The Graph
Graphics Prim
Microsoft Con

Entire contents Copyright © 1989–199

This manual was written by Brian C. Gle
was designed and typeset by Jackie W. P

No portion of this software or manual m
Crescent Software, Inc.

CRESCENT SOFTWARE, INC.
11 BAILEY AVENUE
RIDGEFIELD, CONNECTICUT 06877-45
(203) 438-5300
FAX: (203) 431-4626
Crescent's SupportBBS: (203) 426-59

Second Printing (1.0a)

The Graphics Workshop

Graphics Primitives for Microsoft Compiled Basic

Entire contents Copyright © 1989–1994 by Brian C. Giedt, Ethan Winer and Crescent Software.

This manual was written by Brian C. Giedt, with contributions by Ethan Winer and Don Malin. This manual was designed and typeset by Jacki W. Pagliaro.

No portion of this software or manual may be duplicated in any manner without the written permission of Crescent Software, Inc.

CRESCENT SOFTWARE, INC.
11 BAILEY AVENUE
RIDGEFIELD, CONNECTICUT 06877-4505
(203) 438-5300
FAX: (203) 431-4626
Crescent's Support BBS: (203) 426-5958
Second Printing (1/94)

LICENSE AGREE

Crescent Softwa
documentation to
purposes only.
prohibited, and a
the purchaser.
distribute progra
the primary purp
being sold or dis
of the Graphics
circumstances.
existing copies of

WARRANTY INF

Crescent Software
In the event that i
that event, Crescen
paid, if notified w
returned with the
meet the adv

CRESCENT SOFT
PRICE. Under
of this product be li
for any damages in

LICENSE AGREEMENT

Crescent Software grants a license to use the enclosed software and printed documentation to the original purchaser. Copies may be made for back-up purposes only. Copies made for any other purpose are expressly prohibited, and adherence to this requirement is the sole responsibility of the purchaser. However, the purchaser does retain the right to sell or distribute programs that contain Graphics Workshop routines, so long as the primary purpose of the included routines is to augment the software being sold or distributed. Source code and libraries for any component of the Graphics Workshop program may not be distributed under any circumstances. This license may be transferred to a third party only if all existing copies of the software and documentation are also transferred.

WARRANTY INFORMATION

Crescent Software warrants that this product will perform as advertised. In the event that it does not meet the terms of this warranty, and only in that event, Crescent Software will replace the product or refund the amount paid, if notified within 30 days of purchase. Proof of purchase must be returned with the product, as well as a brief description of how it fails to meet the advertised claims.

CRESCENT SOFTWARE'S LIABILITY IS LIMITED TO THE PURCHASE PRICE. Under no circumstances shall Crescent Software or the authors of this product be liable for any incidental or consequential damages, nor for any damages in excess of the original purchase price.

Table of Contents

Chapter 1 - Intr

Introduction . . .
Overview . . . .
Starting with Gr
Naming Conven
Functions . . . .
Multiple Screens
Screen Modes .
Different Coordi
Passing Parameta
Standard Code .
Why use COMM
Multi-Tasking M

Chapter 2 - Der

Demonstration Pro
ABOUTPCX.BA
DEMOBOX.BA
DEMOBYTE.BA
DEMOCIR.BA
DEMOCUR.BA
DEMODIGI.B
DEMOEDIT.BAS
DEMOEURO.BAS
DEMOFADE.BAS
DEMOFONT.BAS
DEMOFX.BAS
DEMOGW.BAS
DEMOHERC.BAS
DEMOINTR.BAS
DEMOLTS.BAS
DEMOMENU.BAS
DEMOMOUS.BAS
DEMOMOVE.BAS
DEMOPAL.BAS
DEMOPLMS.BAS
DEMOPULL.BAS
DEMOSAVE.BAS
DEMOSCRL.BAS

Chapter 1 - Introduction

Introduction 1-1
Overview 1-1
Starting with Graphics Workshop 1-2
Naming Conventions 1-3
Functions 1-4
Multiple Screens 1-4
Screen Modes 1-5
Different Coordinate Systems 1-6
Passing Parameters 1-8
Standard Code 1-9
Why use COMMON.GW? 1-11
Multi-Tasking Menus 1-13

Chapter 2 - Demonstration Programs

Demonstration Programs 2-1
ABOUTPCX.BAS 2-1
DEMOBOX.BAS 2-1
DEMOBYTE.BAS 2-1
DEMOCIRC.BAS 2-1
DEMOCURS.BAS 2-1
DEMODIGI.BAS 2-2
DEMOEDIT.BAS 2-2
DEMOEURO.BAS 2-2
DEMOFADE.BAS 2-2
DEMOFONT.BAS 2-2
DEMOFX.BAS 2-2
DEMOGW.BAS 2-3
DEMOHERC.BAS 2-3
DEMOINTR.BAS 2-3
DEMOLTS.BAS 2-3
DEMOMENU.BAS 2-3
DEMOMOUS.BAS 2-3
DEMOMOVE.BAS 2-4
DEMOPAL.BAS 2-4
DEMOPLMS.BAS 2-4
DEMOPULL.BAS 2-4
DEMOSAVE.BAS 2-4
DEMOSCRL.BAS 2-5

DEMOSCRN.BAS 2–5
DEMOSHAD.BAS 2–5
DEMOVERT.BAS 2–5
FONT64.BAS 2–5
GWDEMO.BAS 2–5
QSEGUE.BAS 2–6
SCRNDUMP.BAS 2–6
VIEWPCX.BAS 2–6

Assembly Routines

Section 1:

PCX Files and Palettes 3–1
DispPCXC 3–2
DispPCXH 3–3
DispPCXVE 3–4
DispPCXV256 3–5
GetPaletteVGA% 3–6
GetPalTripleVGA 3–7
GWFileSize& 3–8
OpenPCXFile% 3–9
PCXArrayC 3–11
PCXArrayH 3–12
PCXArrayVE 3–13
PCXArrayV256 3–14
PositionPCXVE 3–15
QBinaryLoad% 3–16
SavePCXC 3–18
SavePCXH 3–19
SavePCXVE 3–20
SetPaletteEGA 3–21
SetPalTripleVGA 3–22

Section 2:

BASIC Graphics Replacements 3–24
CircleVE 3–25
ClsVE 3–27
DrawPointH 3–28
DrawPointVE 3–29
DrawPointVEOpts 3–30
GetPointH% 3–32
GetPointVE% 3–33

GPrint0C2 3-34
GPrint0H 3-35
GPrint0V256 3-36
GPrint0VE 3-37
GPrint1VE 3-38
GPrint2VE 3-40
GPrint3V256 3-41
GPrint3VE 3-43
LineBF2VE 3-45
LineBFVE 3-46
LineBVE 3-47
LineStepVE 3-48
LineToVE 3-50
LineToStepVE 3-51
LineVE 3-52
Screen0 3-53
Screen1 3-54
Screen2 3-55
Screen3 3-56
Screen7 3-57
Screen9 3-58
Screen11 3-59
Screen12 3-60
Screen13 3-61

Section 3:

General Screen Manipulation Routines 3-62
ClearScreenArray 3-63
ClearVE 3-64
DrawByteVE 3-65
DrawByteVEOpts 3-67
Fade2EGA 3-69
FadeEGA 3-71
GetCharacter% 3-73
GetLastXCoord% 3-74
GetLastYCoord% 3-75
GetScreenMode% 3-76
GMove1VE 3-77
GMove2VE 3-79
GMove3VE 3-81
GMove4VE 3-83
GScrollVE 3-85

MakeAddressVE% 3-86
MultMonitor% 3-87
PaintBits 3-88
PaintByte 3-90
ScrnDump2 3-92
SetDestPage 3-95
SetGWPages 3-96
SetLastCoord 3-97
SetSourcePage 3-98
SlideDown 3-99
SlideLeft 3-100
SlideRight 3-101
SlideUp 3-102
SplitHorizontal 3-103
TransferEGA 3-104

Section 4:

Mouse Routines 3-106
Using a Mouse with a graphics mode application 3-106
ButtonPress 3-108
GetCursor 3-109
GrafCursor 3-110
HideCursor 3-111
InitMouse 3-113
Motion 3-114
Mouse 3-115
SetCursor 3-116
ShowCursor 3-117

Section 5:

Routines from QuickPak Professional and P.D.Q. 3-118
AltKey% 3-119
FindLast% 3-120
HercThere% 3-121
InStat% 3-122
PDQTimer& 3-123

Chapter 4: BASIC Routines

CircleBAS 4-2
CopyImage 4-3
Digitize 4-4

DisplayBox 4–5
DisplayBoxFill 4–6
DisplayPCXFile 4–7
DisplayPCXFile2 4–8
DoSegue1 4–9
DoSegue2 4–10
DoSegue3 4–11
DoSegue4 4–13
DoSegue5 4–14
DoSegue6 4–15
Draw3DButton 4–16
DrawCursor 4–17
DrawText 4–18
FullZoom 4–19
GEditor 4–20
GetCountLocation% 4–21
GetMouseCount% 4–22
GetOutlineWidth% 4–23
GetWidth% 4–24
GPaintBox 4–25
HandlePCXPalette 4–26
Interlude1 4–27
Interlude2 4–28
LineBAS 4–29
LoadFont 4–30
LoadOutlineFont 4–31
LtsMenuG 4–32
Lts2MenuG 4–33
MsgBoxG 4–34
NightFall 4–36
OutlineText 4–37
PCXCAP 4–38
PositionBox 4–39
PullDownG 4–40
PullDnMSG 4–42
RandomFade 4–44
SetGPFont 4–45
SetGPSpacing 4–46
SetGWFont 4–47
SetGWSpacing 4–48
SetVideo 4–49

ShadeH 4–50
ShadeHorizontal 4–51
ShadeV 4–52
ShadeVertical 4–53
StepText 4–54
VertMenuG 4–55
WhichPCXScreen 4–57
Other Files on the Disk 4–58

Chapter 5: QuickSegue

About The Script Language 5–1
Details About The Script Language 5–1
CLEAR buffer 5–1
INTERLUDE number "anystring" 5–2
LOAD "filename" buffer 5–2
LOCATE horizontal vertical 5–2
SEGUE seguetype subtype color delay 5–3
SEND buffer 5–3
PAUSE 5–3
Segue Types 5–4

Chapter 6: Vector Fonts

Using the Vector Font Editor 6–2
Using the Menu System 6–2
Detailed Function Description Of Menu Items 6–3
Items on the Files menu 6–3
Items on the Edit menu 6–4
Items on the Draw Menu 6–5
Items on the Help Menu 6–6
Using Vector Fonts With Your Program 6–6

APPENDICES

Appendix A

A PCX Primer A–1
Header portion A–1
Data Portion A–2

Appendix B:

The Palette B–1
How The Palette Works B–1
How The EGA Stores Palettes B–1
How The VGA Stores Palettes B–2
What Can You Use Palettes For? B–2

Appendix C:

The GPDat%() Array From GraphPak C–1
Description of the GPDat%() Array C–1
Elements Used By Graphics Workshop C–2
Detailed Listing Of Elements Used In Graphics Workshop C–3

Appendix D:

Converting From QuickPak or GraphPak Professional D–1
Converting from QuickPak Professional D–1
Combining with GraphPak Professional D–3

Appendix E:

Improving Pixel Access Using A Cache Buffer E–1
GetCacheVE% E–3
ReDrawVE E–4
ResetCache E–5

Chapter 1: Introduction

INTRODUCTION

The Graphics Workshop contains the low level graphics routines needed for today's interactive, graphics-based applications. These routines are designed for use with Microsoft QuickBASIC 4.x and BASIC 7. There are three key components to Graphics Workshop:

1. Assembly language routines that provide a dramatic improvement over what is possible by using BASIC alone. Some of the routines improve on BASIC's speed and code size, while others have significant capabilities not available in BASIC.
2. BASIC subprograms and functions that serve as examples and perform a variety of chores that would be tedious or difficult to write yourself.
3. This manual, which provides insights on some of the more advanced topics in graphics programming.

All of the programs include heavily commented source code, not only to show how they are used, but also to explain how they work.

Overview

QuickBASIC has always been a simple and easy-to-use language for creating graphics applications. Built into QuickBASIC from the beginning have been the capabilities to create simple graphics on most of the currently popular display types. It has the ability to draw lines, circles, manipulate graphics images and display text in graphics mode. While these basic abilities are adequate for simple graphics applications, QuickBASIC doesn't have the ability to display pre-created images, such as PCX files created in PC PaintBrush. QuickBASIC doesn't give you the ability to scroll a region of the graphics screen in any direction. QuickBASIC doesn't have the ability to draw lines using one of the arithmetic operations such as XOR. The XOR technique operates by retaining the old information on the screen so that a successive XOR operation will replace the original image on the screen.

Among the deficiencies found in QuickBASIC for graphics applications are the following: Although QuickBASIC can print text in the graphics modes, the process is slow and a programmer cannot specify the background color of the text. Some of the other graphics routines QuickBASIC provides are also slow, particularly reading and writing a single pixel. In terms of code size, if you have ever tried to create an .EXE version of a program using QuickBASIC graphics routines, you will find that it turns into a very large program.

Should all these limitations make you think twice about using QuickBASIC for your graphics programs? The answer is "No". Graphics Workshop supplies you with the capabilities QuickBASIC lacks. Graphics Workshop has the ability to display PCX files created from Hercules, CGA, EGA, and VGA graphics modes. Graphics Workshop gives you a routine to scroll any region of the screen in any direction you choose. Graphics Workshop gives you a line routine that is faster than QuickBASIC's LINE statement and also has the ability to specify XOR, OR, AND, operators or simply a straight replacement (the BASIC LINE statement allows only straight replacement). Graphics Workshop gives you a way to print strings to the screen in any foreground and background color, and at up to 10 times the speed of QuickBASIC's PRINT statement. Graphics Workshop also gives you equivalent but faster routines to replace QuickBASIC's POINT and PSET, statements. These statements are GetPointVE% and DrawPointVE. In addition, the Graphics Workshop's GetCacheVE% routine uses a simple cache so that it does not always have to go to video memory to get the color of the pixel on the screen or even use multiplication to find the pixel's location. This provides a dramatic increase in the speed of getting a pixel's color off the screen.

Finally, because Graphics Workshop provides low level routines to set screen modes and manipulate pages, you may be able to fulfill all of you graphics needs without using QuickBASIC's SCREEN statement, thereby coming out with much smaller executable programs.

Starting with Graphics Workshop

This manual covers the many important topics you will need to know to effectively use Graphics Workshop. In addition to providing a listing of each routine and its calling syntax, many other details are described in depth.

However, if you are familiar with BASIC programming and want to get started right away, simply start QuickBASIC 4 with the GW.QLB Quick Library like this:

```
QB /L GW.QLB
```

If you are using BASIC 7, start it and specify the GW7.QLB Quick Library as follows:

```
QBX /L GW7.QLB
```

Once your version of BASIC has been started, with the appropriate library you may run any of the Graphics Workshop demonstration programs to quickly see what the various routines do, and how they are called. Most demonstration programs start with the letters DEMO, which makes them

easy to identify from the QuickBASIC editor Files menu. All of the demonstration programs are documented in Chapter 2.

After examining the demo programs, the next step would be to use the routines in a program of your own. Before using the routines, it's advisable to read a few of the important sections of this manual.

- Naming Conventions (Chapter 1)
  This section will help you to understand which routines are meant to be used with which screen modes.
- Multiple Screens (Chapter 1)
  This section talks about video memory and how most video adapters have enough memory for two complete graphics screens.
- Different Coordinate Systems (Chapter 1)
  This section talks about the different coordinate systems used by Graphics Workshop, and explains a coordinate system used by Graphics Workshop with which not everyone is familiar.
- Passing Parameters (Chapter 1)
  This section explains how most of the routines in Graphics Workshop pass information. It explains that including the file GWDECL.BAS will alleviate any problems.
- Standard Code (Chapter 1)
  This section goes over the standard code which should be added to your programs to use the different routines in Graphics Workshop.
- The GPDat%() array (Appendix C)
  This section goes over an array that is used to share information between routines. It explains the different elements of the array and how their values can be utilized.

If you are already a Crescent Software customer and are using either QuickPak Professional or GraphPak and want to add Graphics Workshop routines to your existing programs, we suggest you read:

- Converting from QuickPak or GraphPak (Appendix D)
  This section talks about the differences and what is required in converting from one system to another.

Naming Conventions

In this manual and in all of the routines there is a standard naming convention used to specify the screen mode with which the routine is to

be used. A majority of the routines work for both the VGA and EGA high-resolution screen modes. These routines have names like GMove1VE and GPrint0VE. You'll notice that both routines have the letters VE at the end of the routine name. This specifies that the routine is used for the VGA and EGA high-resolution screen modes. The routines which work for only the EGA screen modes have the letters EGA at the end of the routine name. Those routines which work only with VGA 256-color mode have the letters V256 at the end of the routine names.

Functions

An interesting capability of QuickBASIC 4 and BASIC 7 not present in earlier versions of the BASIC compiler is that user-defined functions can execute statements not related to calculating the result of the function.

Beginning with QuickBASIC 4 and BASCOM 6, functions may also be written in assembly language. In the past, the only way an assembler routine could return information was to pass it a variable, and then examine the variable when the routine finished.

Now, however, assembly routines may return a value directly. This feature is used extensively in Graphics Workshop in those cases where returning a value is appropriate. It is important to understand that assembler functions must be declared before they can be used. For example, consider the HercThere% function which checks to see if the MSHERC.COM or QBHERC.COM is resident in memory:

```
IF NOT HercThere% THEN PRINT "MSHERC not loaded"
```

If the function hasn't first been declared, BASIC would have no way to know that HercThere% isn't simply an integer variable.

All of the DECLARE statements you will need for routines contained in Graphics Workshop are in the file GWDECL.BAS. Including GWDECL.BAS does not increase the size of your compiled program.

Multiple Screens

All IBM-compatible graphics hardware, the Hercules, CGA, EGA, and VGA display cards, have both text and graphics modes. Text mode is the normal mode of operation for these cards and supports only text output. Graphics mode is an optional mode that supports the use of both bit-mapped and object graphics operations. Some graphics modes have the ability to store more than one complete graphics screen in memory. Each complete individual screen is considered a page of video memory and each page is assigned a distinct screen number starting with page 0. Page 0

starts out as the visible or foreground screen. Any other pages can be considered background screens as they are not visible at this time. It is not necessary to draw on the visible page; drawing operations can be performed on any of the video pages. QuickBASIC maintains both a visible page and an active page. The visible page is the currently displayed screen. The active page is the page with which all the BASIC graphics statements work whenever they are executed. Using this simple but little-known feature of QuickBASIC, screens can be drawn in the background and then displayed instantaneously. The active page can be set as a parameter of the BASIC SCREEN statement.

In all modes on the Hercules, CGA and EGA graphics adapters, there is sufficient memory to store at least two graphics screens. VGA high-resolution graphics modes, however, don't have enough memory for two graphics pages. Early VGA adapters had only 256K of memory, which simply isn't enough memory to store two of its high resolution screens. One high resolution screen requires 640 x 480 x 4 bits to store the information. That's 80 x 480 x 4 bytes, or 132K. Two screens would then require 264K, which exceeds the memory a VGA adapter has to work with. Unfortunately, in the PC world programmers are required to work for a least common denominator. This means that even though the hardware designers fixed their mistake and later released VGA cards with more memory, there is a significant number of VGA cards which simply are not capable of two video pages.

Screen Modes

QuickBASIC uses the SCREEN statement to set the appropriate graphics mode. The format for the screen statement is as follows:

```
SCREEN Mode, ColorSwitch, ActivePage, VisualPage
```

The Mode value can be selected from the following table:

MODE	MONITOR	SCREEN SIZE	COLORS	PALETTE
0	Color	Text (80x25)		
1	CGA	320x200	4	2 sets
1	CGA	320x200	4	1 set
	(mode 1 with a ColorSwitch of 1)			
2	CGA	640x200	2	16
8	EGA	320x200	16	64
10	EGA	640x200	16	64
7	EGA	640x350	2	64
9	EGA	640x350	16	64
11	VGA	640x480	2	256,000
12	VGA	640x480	16	256,000
13	VGA	320x200	256	256,000

All references to 256,000 colors made in this manual actually refer to 262,144 different colors which are available.

Hercules Graphics Mode is not a supported standard in the IBM PC BIOS. QuickBASIC provides support using a TSR (terminate and stay resident) program. A Mode value of 3 is used when setting the Hercules graphics mode.

Graphics Workshop has a replacement routine for each of these screen modes, including a Hercules screen statement that does not require a TSR program.

When programming you can set the visual page to other than the active page, and then after you have drawn everything in the invisible background, change the visible page. This gives the appearance of drawing complex screens instantaneously.

Different Coordinate Systems

The most common coordinate system is that used by conventional text screens. These screens have 80 horizontal coordinates and 25 vertical coordinates. When you specify a coordinate, you use a number between 1 and 80, or 1 and 25 respectively.

Graphics screens have many more coordinate possibilities. The VGA screen has a resolution of 640 by 480 pixels. When specifying a coordinate

on an VGA graphics screen, the values range from 0 to 639, and from 0 to 479. Charts and graphs usually start from some center point. This point appears at 0, 0 on the number line. To keep graphing simple, graphics mode screens use the upper-left corner of the screen (position 0, 0) as the base coordinate.

If you've tried working with both graphics and text, you'll notice that graphics modes run slower. This is to be expected. It's not that people create less efficient code for graphics mode programs. Rather, in graphics mode there is considerably more work done to place information on the screen. In addition, there are simply more bytes of information to manipulate. A typical character mode screen is an 80x25 matrix or approximately 2,000 (2K) bytes. A typical high resolution VGA screen has 640x480x4 bits, or 132K. It takes more CPU power, and thus more time, to manipulate 132K as opposed to 2K.

To make it simple to understand which type of coordinate system - text or graphics - we are dealing with in the Graphics Workshop routines, we are going to define the variable names that refer to each system. For text mode coordinate systems we will call the variables Row% and Col%. Any variables like Row1% and Cols% also refer to this system. This doesn't mean that Graphics Workshop uses text mode, but it does have the ability to use the text mode coordinate system while running in graphics mode. To talk about a graphics coordinate system, we will use variables like XPos%, Y1%, XPixels%, or YPixels%.

The graphics system of an EGA has 224,000 different pixels on the screen. To change every pixel on the screen would take an exorbitant amount of time. The organization of EGA memory stores 8 pixels in every location of memory. If we could generalize what we want to do with the pixels we could work with these 8 pixels at a time. The 8 pixels are all in a row horizontally on the screen. This allows for 80 groups of these 8 pixels on a row. On a text screen there are 80 columns. This makes it possible to mix coordinate systems. You might ask, "Why mix coordinate systems?" The answer comes from understanding how a computer stores graphics memory. The advantage in working with a screen 8 pixels at a time is, of course, speed. For an example of a major graphics program which mixes coordinate systems, look at either Windows or Presentation Manager. If you create a DOS window and place it in a re-sizable window on the screen you will notice that you can't move it just one or two pixels to the left or right, though you can move it any number of lines up or down. Microsoft could have made it possible to position it at any pixel, but they chose this method to preserve speed.

Similarly, Graphics Workshop has many routines designed to take advantage of this speed as well. This mixed system will use text mode columns and graphics mode lines for its coordinates. The routines which use this new system will use variables like Cols% and Lines% to define the region. Graphics modes using this mixed system will have 80 horizontal coordinates. The EGA high-resolution mode has 350 lines. Similarly the VGA high-resolution mode has 480 lines. It is important to understand that the 80 column resolution is just for positioning. You can still have a different pixel color anywhere on the screen.

Each of the different coordinate systems has been placed in the file GWDECL.BAS. Each of the coordinate systems has been given a name relating to the types in this file. We will refer to the text mode coordinate system as system 0. Then the graphics mode system will be called system 1. The last system is the mixed system with text columns and pixel lines. This system will be called system 2. There are two TYPE definitions for each system. One is the type which holds the coordinates for a single spot. This is called Coord0, Coord1, or Coord2. The other is a variable type which defines a window on the screen. This is called Window0, Window1, or Window2. There is also a definition for a Window1D and a Window2D which will be described below.

There are two ways to describe a window (box) using these coordinate systems. The first is probably more familiar as it defines the window with absolute coordinates for the upper-left and lower-right corners of the window. The second window system uses the coordinate of the upper-left corner of the box and the width and height of the box to define a window. The width and height can be referred to as being delta values from the upper-left corner to the lower-right corner of the region. This coordinate system allows low-level assembly routines to skip the step of calculating how many locations on the screen exist between one side of the window and another. We will refer to these window systems as Window1D and Window2D. The "D" stands for the delta values which it utilizes.

Passing Parameters

The routines in Graphics Workshop were designed with speed in mind. It is for that reason that many of them pass parameters using the BYVAL keyword in QuickBASIC. When a DECLARE statement is made, certain parameters can be preceded by the BYVAL keyword. For an explanation of how this can make a program faster, take a look at the DEMOBNCH.BAS program which demonstrates the speed differences.

QuickBASIC doesn't automatically understand that a routine has parameters which are meant to be passed using the BYVAL keyword. But

if you include the file GWDECL.BAS at the beginning of EVERY one of your programs and modules, you will not have any problems. The GWDECL.BAS file contains DECLARE statements for every assembly routine in Graphics Workshop. Below is an example of how the BYVAL keyword is placed into a DECLARE statement:

```
DECLARE SUB DrawPointVE (BYVAL X%, BYVAL Y%, BYVAL PointColor%)
```

Standard Code

Two forms of standard code will be covered in this section. The first form is for the main program file. The second form is for all of the modules. Below is the standard code for your main program:

```
DEFINT A-Z                          'Makes all variables integers
'$INCLUDE: 'GWDECL.BAS'            'Includes standard declarations
'(Insert other DECLARE statements here)

'$INCLUDE: 'GETVIDEO.BAS'          'Determines monitors attached

'$INCLUDE: 'GPFONT.GW'             'GraphPak Fonts
FontFile$ = "HELV12"
CALL SetGPFont(1)                   'Specify loading font number 1
CALL LoadFont(FontFile$)            'Load font into the Font$() array

'$INCLUDE: 'GWFONT.GW'             'Graphics Workshop Vector Fonts
FontFile$ = "HELV"
CALL SetGWFont(1)                   'Specify loading font number 1
CALL LoadOutlineFont(FontFile$)     'Load font into the
                                    'OutlineFont$() array

CALL SetVideo                       'Sets the screen mode
```

The first line is very important. All of the Graphics Workshop routines expect integer values. The DEFINT statement ensures that all variables are stored and passed in this format. Placing a ! or # symbol after a variable you want to be considered as a floating point number will override the DEFINT for that variable. However, remember that integer math is always faster than floating point math.

As a safeguard, we have provided a BASIC file named GWDECL.BAS that you should include at the very beginning of all your programs and modules. This file contains appropriate declaration statements for all of the Graphics Workshop routines. If you have included this file and then attempt to call a program incorrectly, QuickBASIC will warn you.

Although you don't always have to include all of the above lines of code, the file GWDECL.BAS won't increase the size of your finished program, so it is advisable to include it always. GETVIDEO.BAS determines the monitor you have and dimensions the bare minimum size variables (i.e. 1

element per array) that are necessary for using some of the BASIC routines supplied with Graphics Workshop.

SetVideo is a routine which sets the screen mode. The screen mode is set based upon the value in GPDat%(31). The GPDat%() array is described in fuller detail in Appendix C. You can modify the value in this variable prior to calling SetVideo. You could, for instance, force an EGA hi-resolution mode to be set.

There are two font systems supplied with Graphics Workshop. One is the GraphPak fonts which are small scale fonts. The other is Graphics Workshop's Vector fonts which can be made as large as is possible on the screen. If you are not going to use either of the font systems, you don't need any of the rest of the above lines except the call to SetVideo which is described below. If you are not going to use the Vector (also referred to as Outline) fonts, then don't include the GWFONT.GW file and remove the lines which call LoadOutlineFont. It's as simple as that.

Below is the standard code for all of your modules. When you use the File Load option on the QuickBASIC menu system, it gives you the option to load a module. Any modules loaded in this fashion should utilize the standard code below. There are only three things which should always be added.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'
'(Insert other DECLARE's here)

'$INCLUDE: 'COMMON.GW'
```

Once again it is always best to have a DEFINT statement everywhere. There is no need to use floating point arithmetic when integer arithmetic is much faster. In addition, all of the Graphics Workshop routines expect integers.

GWDECL.BAS should be included at the very beginning of all your programs and modules. This file contains appropriate declaration statements for all of the Graphics Workshop routines. If you have included this file and then attempt to call a program incorrectly, QuickBASIC will warn you. It will add not increase the size of your final compiled programs to have this file included in every module.

COMMON.GW holds BASIC COMMON statements used for sharing variables between routines. The variables shared include font information and the GPDat%() array discussed in Appendix C which is a general purpose array for storing all sorts of useful variables. The following

section describes how to use COMMON.GW and why it should be used in all of your modules.

The GraphPak and Graphics Workshop Vector font systems have the ability to display characters in the extended character set. In character mode applications, the most useful portion of the extended character set seems to be the line drawing characters. In graphics mode, we have a better method of drawing lines: the LineVE routine. So, in graphics mode, the most useful portion of the extended character set is its foreign characters.

The font files EURO.GFN and EURO.QFN work with the GraphPak and Graphics Workshop font systems respectively, and are demonstrated by the DEMOEURO.BAS example program. To enable the European fonts (which requires enlarging the font arrays enough to contain the extended character set), set the variables EuroGPFonts% and EuroGWFonts% to -1 at the locations shown below in the standard code:

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'
'$INCLUDE: 'GETVIDEO.BAS'

EuroGPFonts% = -1           'Set this before reaching the GPFont.GW
                            'include file
'$INCLUDE: 'GPFonts.GW'
CALL SetGPFont (1)
CALL LoadFont ("EURO" )

EuroGWFonts% = -1           'Set this before reaching the GWFont.GW
                            'include file
'$INCLUDE: 'GWFonts.GW'
CALL SetGWFont (1)
CALL LoadOutlineFont ("EURO")
```

Why use COMMON.GW?

When you create a SUB in BASIC, any variables you use there are local to that subroutine. This means that you can use the variable T% even if you are using the same letter T% in some other subroutine or in your main program. This is beneficial in most cases, because you don't have to worry what names you've used elsewhere. But there are times when you want to access variables from other routines within your subroutine. Essentially what you want is a variable name which is global. One way to do this would be to have every routine that needed the variable to have it passed as a parameter. The problem is that if a simple routine was nested inside two, three or even more subroutine calls, each of those routines would need to have the variable passed. Take, for example, the variable Global% which we've created in the example below. It needs to be passed to the routine TestGlobal, but we start in routine A.

```
SUB A (Global%)
        ...
        CALL B(Global%)
        ...
END SUB

SUB B (Global%)
        ...
        CALL C(Global%)
        ...
END SUB

SUB C (Global%)
        ...
        CALL TestGlobal(Global%)
        ...
END SUB
```

As you can see a lot of extra, and unnecessary, passing of information is taking place. If all that needed to be passed was a single variable it would be acceptable, but if it was eight arrays filled with font information, and global information about the screen mode, it would be burdensome. It's hard to imagine adding eight parameters to every routine, not to mention how hard it would be as a programmer to get anything accomplished.

Luckily, QuickBASIC has the answer. It's called the COMMON statement. It allows you to make certain variables global. Its only restriction is that it must be placed in every module within your program. Also, it requires that it sees the same set of COMMON statements in every module of your program. That would be a great deal of typing, which is why we've created COMMON.GW. COMMON.GW is a BASIC include file which you must place at the top of all of your routines should you decide to use either of the font systems within Graphics Workshop, or if you use the GPDat%() array inside your program. Note that the GPDat%() array is created automatically inside the include file GETVIDEO.BAS, shown in the "Standard Code" section in this chapter. Shown below are the contents of the COMMON.GW file. If you're already using COMMON statements of your own, we suggest you combine the two sets of COMMON statements into one include file.

```
COMMON SHARED GPDat%(), Font$(), FontWidth%(), FontHeight%()
COMMON SHARED OutLineFont$(), FontSize%(), OutLineHeight%()
COMMON SHARED Tile$(), AltTile$()
```

The keyword SHARED should also appear on each line as shown above. Note, the COMMON.GW include file is just part of what we have recommended you place at the top of every one of your modules. See the "Standard Code" section for more details.

Multi-Tasking Menus

One of the more exciting capabilities we have provided in Graphics Workshop is a system of menus that can run in the graphics modes. These routines create menu systems that appear to be running just as if they were running in the text modes. A special shadow feature has been duplicated using an advanced routine which can change many pixel colors at a time. This shadow effect can be turned off by commenting out the two lines which call the routine GPaintBox.

These menus have a multi-tasking feature which allows your program to be doing things while the menu system is waiting for the user to make a selection. Where most menu programs simply sit in an idle loop waiting for a key press, both the pull-down and vertical menu subprograms let you continue your program if you wish to do so.

To accomplish this, an "Action" parameter (Action%) has been added to the calling sequence. Depending on the setting of this variable, the menus can be instructed to operate in a number of different ways.

The Action variable has six different possible settings which tell PullDown-G and VertMenuG how they are to behave. Each of the possible Action values is described in detail below.

If Action is set to zero, the menu will operate the way you would expect a "normal" menu to work. The underlying screen is first saved, then the menu is displayed, and finally an INKEY$ loop repeatedly waits for the user to press a key or a mouse button. Once a key or a mouse button has been pressed, the original screen is restored and control is returned to the calling program. The Choice variable(s) may then be examined to see what selection the user chose.

When Action is set to 1, both PullDownG and VertMenuG save the screen and display themselves. Control is then returned to the calling program immediately and Action will be set to 3 for subsequent calls. Since Action 3 is how you will be polling the menu subsequently, this saves you an extra step.

Setting Action to 2 lets you re-display the menu in those cases where it may have been overwritten by another possibly overlapping menu. Action 2 also resets itself to 3 for subsequent calls.

If the menus are called with Action 3, the keyboard and mouse are merely polled to see if a key or button has been pressed. If Action is still set to 3 when the menu returns, it means that no keys or mouse buttons were

pressed. If Action is returned set to 4, the user either made a selection or pressed Escape. In this case, the Choice, Menu, and Ky$ variables should be examined. Look at the programs DEMOPULL.BAS and DEMOVERT.BAS to see how these variables can be examined.

The last Action value is 5 which tells VertMenuG or PullDownG to remove itself and restore the original screen.

If you intend to create stacked menus, you should be aware of one important point. Because each menu saves its own underlying screen, the screen that was saved first will be destroyed when the menu is called again. This means that it is up to you to save and restore each screen in succession manually, except for the last one.

DEMOMENU.BAS provides complete demonstration of using PullDown-G and VertMenuG in a typical programming context.

Chapter 2: Demonstration Programs

DEMONSTR

ABOUTPCX.BAS

ABOUTPCX.BAS
header. It is an e
particular monitor
size, and suggests

DEMOBOX.BAS

DEMOBOX.BAS
boxing routine for
PositionBox, its de
using the cursor ke
for one of two co
those sources, use

DEMOBYTE.BAS

DEMOBYTE.BAS
pixels, draw patte
The DrawByte VE
the same ef
routine can be a

DEMOCIRC.BAS

DEMOCIRC.BAS
Workshop. Use thi
when creating prog
when you want to u
Also, the CircleVE
your programs will

DEMOCURS.BAS

DEMOCURS.BAS
in CURSOR.BAS
the position you spec
or not the cursor is
positioning with a

DEMONSTRATION PROGRAMS

ABOUTPCX.BAS

ABOUTPCX.BAS displays all the information contained in a .PCX file's header. It is an excellent tool for testing the use of a .PCX file on a particular monitor. ABOUTPCX tells the version of the file, the image size, and suggests which type of monitor it was created on.

DEMOBOX.BAS

DEMOBOX.BAS shows the operation of the exclusive-oring (XOR) boxing routine for defining a region on the screen. A complete routine, PositionBox, is demonstrated here. This routine accepts keyboard input using the cursor keys to adjust the size of the box. Adjustments are made for one of two corners, upper-left and lower-right. To toggle between these corners, use the SpaceBar.

DEMOBYTE.BAS

DEMOBYTE.BAS shows how to use the DrawByteVE routine to display pixels, draw patterns, simulate the LINE statement in BASIC and more. The DrawByteVE routine can draw up to 8 pixels at a time, but all using the same color. If these requirements fit your needs, then the DrawByteVE routine can be a fast way to draw images.

DEMOCIRC.BAS

DEMOCIRC.BAS shows the use of the CircleVE routine within Graphics Workshop. Use this routine to emulate the BASIC CIRCLE statement when creating programs that don't use the BASIC SCREEN statement, or when you want to use its OR, AND, or XOR logical operation abilities. Also, the CircleVE routine does not require floating point arithmetic so your programs will be smaller.

DEMOCURS.BAS

DEMOCURS.BAS shows the use of the graphics cursor routines contained in CURSOR.BAS. The routine DrawCursor XORs a graphics cursor at the position you specify. A value is maintained that keeps track of whether or not the cursor is currently visible. DrawCursor also demonstrates positioning within a proportional font string.

DEMODIGI.BAS

DEMODIGI.BAS shows the screen digitizing routines in DIGITIZE.BAS. These routines take a portion of the screen and lower the apparent resolution. This can be used to create an artistic effect for the screen it is used on.

DEMOEDIT.BAS

DEMOEDIT.BAS shows how the GEditor routine can provide a complete field editor within graphics mode. The routine uses the DrawCursor routine to implement a graphics mode cursor, and uses the GPrint0VE routine to accomplish printing the text information within the string.

DEMOEURO.BAS

DEMOEURO.BAS shows how to use the GraphPak and Graphics Workshop fonts systems with characters extending through the entire ASCII character set. Many of the characters in the extended character set are used in European languages.

DEMOFADE.BAS

DEMOFADE.BAS shows the FadeEGA routine in action. The FadeEGA routine is the most complex of the graphics transfer routines. FadeEGA brings an image to the visible screen randomly, 4 pixels at a time. The 4 pixels are arranged in the form of a 2x2 square. Two uses of the FadeEGA routine are demonstrated.

DEMOFONT.BAS

DEMOFONT.BAS presents all the fonts available in Graphics Workshop. Both the Vector Fonts described in Chapter 6 and the fonts provided from GraphPak may be drawn at any size, and at any angle. Also demonstrated is the use of multiple fonts on the same screen.

DEMOFX.BAS

DEMOFX.BAS demonstrates the GMove1VE and GMove3VE routines for moving images on and off the screen. The GMove1VE routine is used by BASIC subroutines to slide images onto the screen. The GMove3VE routine is used by a BASIC subroutine to split the image and move to the

screen in two parts. The GMove1VE routine can also be used to create a mirror image of an image on the screen.

DEMOGW.BAS

DEMOGW.BAS demonstrates many of the routines in Graphics Workshop, and gives a variety of methods for using them.

DEMOHERC.BAS

DEMOHERC.BAS shows the graphics primitives that can be used on a Hercules monitor without requiring the use of the TSR MSHERC.COM.

DEMOINTR.BAS

DEMOINTR.BAS shows the "interludes" available for adding to the QuickSegue demo (QSEGUE.BAS). Two interludes are shown. One uses a movie clicker to display a title. The other provides an interesting background effect which can be placed behind titles to add motion to a static image.

DEMOLTS.BAS

DEMOLTS.BAS demonstrates the two Lotus-style menu programs, Lts-MenuG and Lts2MenuG, which come with graphics workshop. DEMOLTS.BAS demonstrates how to initialize the menu arrays and set up the display to use the routine.

DEMOMENU.BAS

DEMOMENU.BAS shows the use of the PullDownG, VertMenuG, and MsgBoxG routines together as a complete system. The multi-tasking feature of the PullDownG and MsgBoxG routines are implemented here.

DEMOMOUS.BAS

DEMOMOUS.BAS shows the use of mouse routines for graphics mode. It demonstrates the proper use of the HideCursor and ShowCursor routines when programming for mouse support in the graphics modes. It also demonstrates the abilities of all of the other mouse routines provided with Graphics Workshop.

DEMOMOVE.BAS

DEMOMOVE.BAS shows how to use the GMove1VE and GMove2VE routines to duplicate images, simulate the GET and PUT in BASIC with a major speed improvement, and how subroutines can be created to flip images or manipulate images in almost any fashion. The GMove1VE and GMove2VE routines do not require setting up a large array like the GET and PUT statements in BASIC; all work is done entirely in video memory.

DEMOPAL.BAS

DEMOPAL.BAS shows how using the palette can increase the impression of a presentation. It also demonstrates how to simulate motion using the palette. Finally, it demonstrates the need for a proper set of colors when creating a real-life image.

DEMOPLMS.BAS

DEMOPLMS.BAS shows the use of a Windows(tm)-like graphics pull-down menu system. The routine GPrint0VE is a vital part of the PullDnMSG menu system for graphics mode. It allows the text to have a background color, which is not possible with BASIC's PRINT statement while running in graphics mode. Another vital routine is GMove2VE, which is used to save and restore parts of the graphics screen which lie beneath the pull-down menus.

DEMOPULL.BAS

DEMOPULL.BAS shows the use of graphics pull-down menus. The routine GPrint0VE is a vital part of the PullDownG menu system for graphics mode. It allows the text to have a background color, which is not possible with BASIC's PRINT statement while running in graphics mode. Another vital routine is GMove2VE, which is used to save and restore parts of the graphics screen which lie beneath the pull-down menus.

DEMOSAVE.BAS

DEMOSAVE.BAS shows the use of SavePCXVE for storing a graphics screen into the .PCX file format. The routine SavePCXVE is written in assembly and is therefore very fast.

DEMOSCRL.BAS

DEMOSCRL.BAS shows the GScrollVE routine in action. Four windows are scrolled simultaneously with and without a delay. In the next portion of the demo, four items placed on the screen are moved inward toward each other to form one object in the middle.

DEMOSCRN.BAS

DEMOSCRN.BAS shows how the KeepData% parameter available with all of the Graphics Workshop Screen mode routines can be used when swapping between graphics mode to text mode and back again.

DEMOSHAD.BAS

DEMOSHAD.BAS shows the ShadeVertical and ShadeHorizontal routines that provide a flowing transition of color to any window. Use these routines to add a background to demonstration images or titles.

DEMOVERT.BAS

DEMOVERT.BAS shows the use of graphics vertical scrolling menus. The routine GPrint0VE is a vital part of the VertMenuG menu system for graphics mode. It allows the text to have a background color, which is not possible with BASIC's PRINT statement while running in graphics mode. Another vital routine is GMove2VE, which is used to save and restore the graphics screen which lies beneath the vertical scrolling menus.

FONT64.BAS

FONT64.BAS is a font editor written in QuickBASIC to create vector fonts. It demonstrates use of the pull-down menu system for graphics, and provides an excellent way to create new fonts. It accepts GraphPak fonts as a base font which you can outline to form new fonts.

GWDEMO.BAS

GWDEMO.BAS is the demo program the Graphics Workshop demo disk. It includes an example of "mousable" three-dimensional buttons, and introduces some different fade types. Benchmarks are also shown comparing the speed improvements over BASIC's equivalent routines.

QSEGUE.BAS

This program called QuickSegue is a complete graphics slide show program. It accepts a script consisting of commands from an ASCII input file, used to load graphics files and send them to the screen. The QuickSegue script language is described in detail in Chapter 5.

SCRNDUMP.BAS

SCRNDUMP.BAS shows the print routine ScrnDump2 in action. This routine can print any graphics screen to either an EPSON or compatible dot-matrix, or to a Hewlett Packard LaserJet in either Portrait or Landscape mode. If you're printing to the LaserJet, there is also an option to perform rudimentary scaling.

VIEWPCX.BAS

VIEWPCX.BAS loads a .PCX graphics file, determines the screen mode used to store the screen (EGA, VGA, CGA, or HERC), and then sends the graphics file to the screen. Pressing any key returns to DOS. The program also reads the palette information from the .PCX file and adjusts the palette accordingly.

Chapter 3: Assembly Routines

■ ■ ■

Virtually all of the
support macros
the segment
otherwise would
GW.VARS.ASM.
variables used by
in GW.LIB and
with Graphics

SECTION 1: PCX

A PCX file is a
PaintBrush. The
PCX images. Int
Appendix A. Info
in Appendix B.

Virtually all of the assembly routines rely on some support material. The support material is generally shared data locations in memory. Data like the segment to use for the proper video page, and local data which otherwise would be duplicated for each routine are contained in the file GWVARS.ASM. The assembler file GWVARS.ASM holds the common variables used by most of the routines in this chapter. This file is contained in GW.LIB and GW7.LIB and will be required by most programs created with Graphics Workshop.

SECTION 1: PCX FILES AND PALETTES

A PCX file is a graphics file format created by ZSoft, the makers of PC PaintBrush. The routines in this section are used to access and display PCX images. Information about the PCX file format is contained in Appendix A. Information about the Palette and how it is used is contained in Appendix B.

DispPCXC

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

DispPCXC continues the loading process started by OpenPCXFile% and displays the image to a CGA specified video page.

■ **Syntax:**

```
CALL DispPCXC (BYVAL VideoPage%)
```

■ **Where:**

VideoPage% is 0 for the default first display page (Visual Display Page). A value of 1 specifies the second display page.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The function OpenPCXFile% must be called first, as it opens the PCX file and loads the header information to determine which screen mode the PCX file is intended for.

Due to the nature of the CGA's video memory, this routine will work on both the CGA 2-color graphics mode and the CGA 4-color graphics mode. Loading a CGA 4-color graphic image while the computer is in the CGA 2-color mode, will translate the image into a tiled 2-color image.

Use of this routine is demonstrated in the VIEWPCX.BAS example program and the BASIC routine DisplayPCXFile.

■ **See Also:**

OpenPCXFile%

DispPCXH

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  DispPCXH continues the loading process started by OpenPCXFile% and displays the image to a Hercules specified video page.

- **Syntax:**

```
CALL DispPCXH (BYVAL VideoPage%)
```

- **Where:**

  VideoPage% is 0 for the default first display page (Visual Display Page). A value of 1 specifies the second display page.

- **Comments:**

  All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

  The function OpenPCXFile% must be called first, as it opens the PCX file and loads the header information to determine which screen mode the PCX file is intended for.

  The Hercules screen mode is of an odd size, 720x348 pixels when created with the use of MSHERC.COM. You can replace the BASIC SCREEN 3 statement in your program with the Screen3 routine provided with the Graphics Workshop to set the Hercules into graphics mode. This method does not require the use of the TSR program MSHERC.COM, yet it affords full compatibility.

  Use of this routine is demonstrated in the VIEWPCX.BAS example program and the BASIC routine DisplayPCXFile.

- **See Also:**

  OpenPCXFile%

DispPCXVE

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  DispPCXVE continues the loading process started by OpenPCXFile% and displays the image to a VGA or EGA specified video page.

- **Syntax:**

```
CALL DispPCXVE (BYVAL VideoPage%)
```

- **Where:**

  VideoPage% is 0 for the default first display page (Visual Display Page). A value of 1 specifies the second display page.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The function OpenPCXFile% must be called first, as it opens the PCX file and loads the header information to determine which screen mode the PCX file is intended for.

The DispPCXVE routine works equally well when BASIC is operating in SCREEN 7, 8, 9, 11, and 12 since the video memory for all of these modes is identical. This routine works for all the screen modes which utilize the plane system created for EGA and VGA graphics. The EGA and VGA 2-color graphics modes do not utilize the plane scheme most of the other EGA and VGA screens use. The DispPCXVE routine will still work equally as well on those. Also, note that PaintBrush for Windows version 2 saves only three of the four graphics planes in the file (leaving out the intensity plane). This routine will load those files properly as well.

Use of this routine is demonstrated in the VIEWPCX.BAS example program and the BASIC routine DisplayPCXFile.

- **See Also:**

  OpenPCXFile%

DispPCXV256

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

DispPCXV256 continues the loading process started by OpenPCXFile% and displays the image to a VGA 256-color mode specified video page.

■ **Syntax:**

```
CALL DispPCXV256 (BYVAL VideoPage%)
```

■ **Where:**

VideoPage% is 0 for the default first display page (Visual Display Page). A value of 1 specifies the second display page.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The function OpenPCXFile% must be called first, as it opens the PCX file and loads the header information to determine which screen mode the PCX file is intended for.

Use of this routine is demonstrated in the VIEWPCX.BAS example program and the BASIC routine DisplayPCXFile.

■ **See Also:**

OpenPCXFile%

GetPaletteVGA%

Assembler
function contained in GW.LIB

- **Purpose:**

  GetPaletteVGA% returns the current value of the palette register specified. This routine is used for the EGA and VGA screen modes, but only if you are using a VGA video adapter.

- **Syntax:**

```
Colr% = GetPaletteVGA%(BYVAL PalRegister%)
```

- **Where:**

  PalRegister% is one of the 16 available colors for the EGA screen modes (use 0 through 15).

  Colr% is the value returned and is between 0 and 63 which represents the color value from the palette.

Comments:

Because GetPaletteVGA% has been designed as a function and passes a parameter by value, it must be declared before it can be used. Including the file GWDECL.BAS at the beginning of all your programs will avoid any problems as it contains a declaration for this function.

The short example below shows how to get the palette value of color 1 on a VGA monitor.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets monitor in a VGA mode

PaletteValue% = GetPaletteVGA%(1)
```

The EGA video registers are write-only. There is no way to read the EGA registers which contain the palette information. Therefore, getting an EGA palette value is not a simple task. There are two options to determine the current palette settings. One, if you are setting the palette registers, you can maintain a list of what values you have set. Then you will know what is the value of each palette register. Two, there was an excellent article in the March 1990 issue of Programmer's Journal which discussed a method of re-mapping the EGA's palette.

- **See Also:**

  SetPaletteEGA

GetPalTripleVGA

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

GetPalTripleVGA returns the Red, Green, and Blue values which make up the specified palette. This routine can be used only in the VGA screen modes.

■ **Syntax:**

```
CALL GetPalTripleVGA(BYVAL PalRegister%, Red%, Green%, Blue%)
```

■ **Where:**

PalRegister% specifies one of the 256 available colors for the VGA screen modes (use 0 through 255), or one of the 16 available colors for the higher resolution screens (use 0 through 15).

Red%, Green%, and **Blue%** return values between 0 and 63 for each of the color planes for that palette register.

Comments:

One parameter for this routine is passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

To get the palette information for the color 3 on a VGA screen mode, use the following code fragment:

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 13                 'sets monitor in a VGA mode

GetPalTripleVGA(3, Red%, Green%, Blue%)
```

Palette values read from .PCX files are shifted two bits to the left. The routine HandlePCXPalette does an integer divide by 4 to shift the values two bits to the right.

■ **See Also:**

SetPalTripleVGA, HandlePCXPalette

GWFileSize&

Assembler
function contained in GW.LIB

- **Purpose:**

  GWFileSize& will quickly return the length of a named file.

- **Syntax:**

```
Size& = GWFileSize& (Filename$)
```

- **Where:**

  Filename$ is the name of the file.

  Size& receives its length in bytes. If the file does not exist Size& is instead assigned a value of -1.

Comments:

Because GWFileSize& has been designed as a function, it must be declared before it can be used. Including the file GWDECL.BAS at the beginning of all your programs will avoid any problems as it contains a declaration for this function.

The brief example that follows shows how to find the file size of the file DEMOSHAD.PCX.

```
Size& = GWFileSize& ("DEMOSHAD.PCX")
```

This product's main purpose for using GWFileSize& is to determine the file size of .PCX files. The section on the routine QBinaryLoad% goes into more detail of the use of this routine.

Those users of QuickPak Professional will note that QuickPak has a FileSize& routine. The difference between the two routines is that GWFileSize& does not use the critical error handler that comes with QuickPak Professional.

- **See Also:**

  QBinaryLoad%

OpenPCXFile%

Assembler
function contained in GW.LIB

- **Purpose:**

  OpenPCXFile% opens the specified PCX file, and loads the header information, including palette information, into the string specified.

- **Syntax:**

```
Array$ = SPACE$(68 + 768)
Success% = OpenPCXFile%(Filename$, Array$)
```

- **Where:**

  Filename$ is a string containing the filename of the PCX file.

  Array$ is a string of length (68 + 768). The first 68 bytes receive the header information. If the file is a 256-color PCX file, then the information contained in the last 768 bytes of this string are the palette information for the 256-color mode.

Comments:

After opening the PCX file with this function, one of the following routines can be used to display the PCX file: DispPCXC, DispPCXH, DispPCXVE, DispPCXV256.

The following code fragment will load and display a PCX file whose name is contained in Filename$. It assumes that the correct screen mode for the PCX file has already been set.

```
Array$ = SPACE$(68 + 768)
IF NOT OpenPCXFile%(Filename$, Array$) THEN EXIT SUB

CALL WhichPCXScreen(Array$, WhichScreen%)

CALL HandlePCXPalette(Array$, WhichScreen%)

IF WhichScreen = 4 OR WhichScreen = 6 THEN
        CALL DispPCXC(VideoPage%)
ELSEIF WhichScreen = 2 THEN
        CALL DispPCXH(VideoPage%)
ELSEIF WhichScreen = 9 THEN
        CALL DispPCXV256(VideoPage%)
ELSE
        CALL DispPCXVE(VideoPage%)
        CALL ClearVE
END IF
```

Use of this routine is demonstrated in the VIEWPCX.BAS example program and the BASIC routine DisplayPCXFile.

■ **See Also:**

DispPCXC, DispPCXH, DispPCXVE, DispPCXV256, DisplayPCXFile, HandlePCXPalette, WhichPCXScreen

PCXArrayC

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

PCXArrayC takes an array containing a PCX file and sends it to the CGA screen specified by video page.

■ **Syntax:**

```
CALL PCXArrayC (BYVAL ArraySeg%, BYVAL VideoPage%)
```

■ **Where:**

ArraySeg% is the segment of an integer array which already holds the complete .PCX file, starting at the memory location of the first element of the array and extending to the last memory location of the array.

VideoPage% is 0 for the default first display page (Visual Display Page). A value of 1 specifies the second display page.

Comments:

Both parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The Array%() must be loaded with the QBinaryLoad% routine which loads a binary file into a specific memory location. A complete example of using the PCXArrayX routines is shown in context in the comments for the QBinaryLoad% routine.

Due to the nature of the CGA's video memory, this routine will work on both the CGA 2-color graphics mode and the CGA 4-color graphics mode. Loading a CGA 4-color graphic image while the computer is in the CGA 2-color mode will translate the image into a tiled 2-color image.

An example of using this routine is contained in the demonstration program QSEGUE.BAS.

■ **See Also:**

QBinaryLoad%

PCXArrayH

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  PCXArrayH takes an array containing a PCX file and sends it to the Hercules screen specified by video page.

- **Syntax:**

```
CALL PCXArrayH (BYVAL ArraySeg%, BYVAL VideoPage%)
```

- **Where:**

  ArraySeg% is the segment of an integer array which already holds the complete .PCX file, starting at the memory location of the first element of the array and extending to the last memory location of the array.

  VideoPage% is 0 for the default first display page (Visual Display Page). A value of 1 specifies the second display page.

Comments:

Both parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The Array%() must be loaded with the QBinaryLoad% routine which loads a binary file into a specific memory location. A complete example of using the PCXArrayX routines is shown in context in the comments for the QBinaryLoad% routine.

The Hercules screen mode is of an odd size, 720x348 pixels when created with the use of MSHERC.COM. Rather than using MSHERC.COM, you can replace the BASIC SCREEN 3 statement in your program with the Screen3 routine provided with Graphics Workshop to set the Hercules into graphics mode. This method does not require the use of the TSR program MSHERC.COM, but would allow a program to display a PCX file in the Hercules mode. See the routine Screen3 for details and restrictions.

An example of using this routine is contained in the demonstration program QSEGUE.BAS.

- **See Also:**

  QBinaryLoad%, HercThere%, Screen3

PCXArrayVE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

PCXArrayVE takes an array containing a PCX file and sends it to the EGA or VGA screen specified by video page.

■ **Syntax:**

```
CALL PCXArrayVE (BYVAL ArraySeg%, BYVAL VideoPage%)
```

■ **Where:**

ArraySeg% is the segment of an integer array which already holds the complete .PCX file, starting at the memory location of the first element of the array and extending to the last memory location of the array.

VideoPage% is 0 for the default first display page (Visual Display Page). A value of 1 specifies the second display page.

Comments:

Both parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The Array%() must be loaded with the QBinaryLoad routine which loads a binary file into a specific memory location. A complete example of using the PCXArrayX routines is shown in context in the comments for the QBinaryLoad% routine.

The PCXArrayVE routine works equally well when BASIC is operating in SCREEN 7, 8, 9, 11, and 12 since the video memory for all of these modes is identical. This routine works for all the screen modes which utilize the plane system created for EGA and VGA graphics. The EGA and VGA 2-color graphics modes do not utilize the plane scheme most of the other EGA and VGA screens use. The PCXArrayVE routine will still work equally as well on those. Also, note that PaintBrush for Windows version 2 saves only three of the four graphics planes in the file (leaving out the intensity plane). This routine will load those files properly as well.

An example of using this routine is contained in the demonstration program QSEGUE.BAS.

■ **See Also:**

QBinaryLoad%

PCXArrayV256

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  PCXArrayV256 takes an array containing a PCX file and sends it to the VGA screen running in 256-color mode.

- **Syntax:**

```
CALL PCXArrayV256 (BYVAL ArraySeg%, BYVAL VideoPage%)
```

- **Where:**

  ArraySeg% is the segment of an integer array which already holds the complete .PCX file, starting at the memory location of the first element of the array and extending to the last memory location of the array.

  VideoPage% is 0 for the default first display page (Visual Display Page). Memory limitations allow only one page on the VGA when in 256-color mode.

Comments:

Both parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The Array%() must be loaded with the QBinaryLoad% routine which loads a binary file into a specific memory location. A complete example of using the PCXArrayX routines is shown in context in the comments for the QBinaryLoad% routine.

An example of using this routine is contained in the demonstration program QSEGUE.BAS.

- **See Also:**

  QBinaryLoad%

PositionPCXVE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

PositionPCXVE sets the screen location to place the next PCX image on an EGA or VGA high-resolution screen mode. This routine is called after OpenPCXFile.

■ **Syntax:**

```
CALL PositionPCXVE (BYVAL LineStart%, BYVAL ColStart%)
```

■ **Where:**

LineStart% is a value between 0 and 479 for a VGA display.

ColStart% is a column number between 1 and 80.

Comments:

Both parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

Proper usage of this routine is shown in the example subroutine DisplayPCXFile2.

QBinaryLoad%

Assembler
function contained in GW.LIB

■ **Purpose:**

QBinaryLoad% loads the PCX graphics file specified by Filename$ into the integer array so that it can be displayed on the screen at a later time. If the file is read successfully, this function returns a non-zero value.

■ **Syntax:**

```
Success = QBinaryLoad%(Filename$, SEG Array%)
IF NOT Success THEN PRINT "File Error"
```

■ **Where:**

Filename$ is a string containing any valid file name. No wildcards should be used here.

Array%() is an integer array with half as many elements as there are bytes in the file. The array can be one-dimensional or two-dimensional. A two-dimensional array is required if you intend to load files larger than 64K.

Comments:

A method for loading a PCX file to the EGA or VGA high-resolution screen modes:

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'
SCREEN 9

Filename$ = "Graph1.PCX"
Size& = GWFileSize&(Filename$)
REDIM Array%(Size& \ 4 + 1, 1)
Success = QBinaryLoad(Filename$, SEG Array%(0, 0))
IF Success THEN
  ArraySeg% = VARSEG(Array%(0, 0))
  CALL PCXArrayVE(ArraySeg%, 0)
ELSE
  PRINT "Error Loading File"
END IF
```

The above code has been simplified to expect the PCX file to be for the EGA or VGA high-resolution screen modes. To use the PCXArrayC, PCXArrayH, or PCXArrayV256 routines, simply replace the above line containing the call to PCXArrayVE.

Code like that in the routine WhichPCXScreen can be used to determine the screen mode a PCX file was created on. Note that the code in WhichPCXScreen will not work directly, because it is designed for the

PCX display routines like DispPCXVE which do not load the entire file into memory before displaying it. Furthermore, when using the above code fragment for the array method of displaying PCX files, it is likely that the screen mode will be known ahead of time.

■ **See Also:**

GWFileSize&, PCXArrayVE, PCXArrayC, PCXArrayH, PCXArrayV256

SavePCXVE

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  SavePCXVE saves the current EGA or VGA high-resolution screen to the .PCX file specified by Filename$.

- **Syntax:**

```
CALL SavePCXVE(Filename$)
```

- **Where:**

  **Filename$** is a string which holds the name of the .PCX file to be saved. Filename$ must contain the extension ".PCX". If the extension is not used, then the PCX file will not load properly using the DisplayPCXFile routine supplied with Graphics Workshop.

Comments:

This routine determines the current EGA or VGA video mode and adjusts itself to save the appropriate number of video lines.

The short example below demonstrates saving an EGA screen containing a red circle to the file REDCIRC.PCX.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'set the monitor in the EGA mode
CIRCLE (100, 100), 80, 4                  'draw a red circle

CALL SavePCXVE ("REDCIRC.PCX")
```

A demonstration of using this routine can be found in the example program DEMOSAVE.BAS.

The routine SavePCXVE can also be used as a function. Their are declare statements in GWDECL.BAS which need to be changed if you want to use SavePCXVE% as a function. It will return a 0 if the file operation was successful, and a non-zero value otherwise.

- **See Also:**

  DispPCXVE, PCXArrayVE

SetPaletteEGA

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

SetPaletteEGA sets the color value for the palette register specified. This routine should be used only in the EGA screen modes. The SetPalTripleVGA routine supplies the VGA screen modes with a much larger palette from which to choose.

■ **Syntax:**

```
CALL SetPaletteEGA(BYVAL PalRegister%, BYVAL Value%)
```

■ **Where:**

PalRegister% is one of the 16 available colors for the EGA screen modes (use 0 through 15).

Value% is a number between 0 and 63 which is used to represent one of the available colors in the palette.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The code fragment below demonstrates how to change the palette for color number 1 to be represented as white on the screen.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'set the monitor in an EGA mode

CALL SetPaletteEGA (1, 63)                  '63 represents white
```

This routine replaces the PALETTE statement in BASIC for use with the EGA screen modes. If you are not using QuickBASIC's SCREEN statement, then using BASIC's PALETTE statement would cause an error. If you are using the Graphics Workshop Screen9 routine or one of the other Graphics Workshop routines which set the screen mode, use the SetPaletteEGA routine instead of BASIC's PALETTE statement.

■ **See Also:**

GetPaletteVGA, SetPalTripleVGA, Screen9

SetPalTripleVGA

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  SetPalTripleVGA sets the red, green and blue values for a specified color in the palette. This routine can be used only in the VGA screen modes.

- **Syntax:**

```
CALL SetPalTripleVGA (BYVAL PalRegister%, BYVAL Red%, BYVAL _
  Green%, BYVAL Blue%)
```

- **Where:**

  PalRegister% is one of the 256 available colors for the VGA screen modes (use 0 through 255), and one of the 16 available colors for the higher resolution screen modes (use 0 through 15).

  Red%, Green% and **Blue%** are all values between 0 and 63, which specify the intensity of that color in the overall color generated for this palette register.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The code fragment below shows how to set color number 13 to be represented on the screen as white.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor into a VGA mode

CALL SetPalTripleVGA (13, 63, 63, 63)
```

Setting the Red, Green and Blue values to their maximum values (63) will provide an equal mix of all colors at their brightest intensities, and hence produce the color of white.

This routine replaces the PALETTE statement in BASIC for use with the VGA screen modes. If you are not using QuickBASIC's SCREEN statement, then using BASIC's PALETTE statement would cause an error. If you are using the Graphics Workshop Screen13 routine or one of the other Graphics Workshop routines which set the screen mode, use the SetPaletteEGA routine instead of BASIC's PALETTE statement.

■ **See Also:**

SetPaletteEGA, GetPalTripleVGA, Screen13

Section 2:
BASIC Graphics Replacements

The routines in this section could be used to replace graphics statements like BASIC's PSET. Most of the routines in this section are much faster than BASIC's equivalents. These routines also provide more power than their BASIC equivalents.

Note:

There are also equivalent routines for BASIC's PALETTE statement, but these are documented in Section 1 of this chapter.

CircleVE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

CircleVE is a replacement for the BASIC CIRCLE statement that not only doesn't require using floating point arithmetic, but also has the ability to use logical operations like XOR, OR, and AND when drawing the circle.

■ **Syntax:**

```
CALL CircleVE(BYVAL XCenter%, BYVAL YCenter%, BYVAL _
  Radius%, BYVAL Colr%, BYVAL XAspect%, BYVAL YAspect%)
```

■ **Where:**

XCenter% and **YCenter%** define the center of the circle on the screen.

Radius% is the radius of the circle in pixels.

Colr% is the color that the circle will be drawn in. This variable also serves to tell the routine which of the logical operations should be used. Add one of the values in the table below to change the logical operation performed when drawing the circle.

VALUE	LOGICAL OPERATION
0	PSET (Replace)
2048	OR
4096	AND
6144	XOR

Any other values will produce unusual (and probably undesirable) effects.

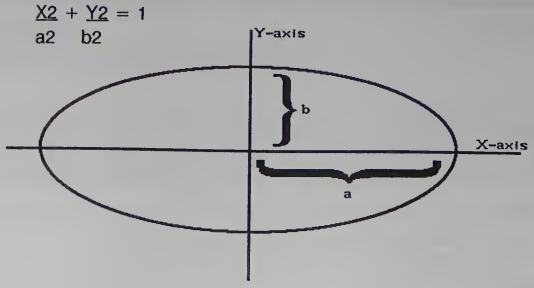
To create a rubber-banding effect with the color grey, you would pass CircleVE the following color value:

```
Colr% = 7 + 6144
```

XAspect% and **YAspect%** are used to draw ellipses.

Comments:

The standard equation for an ellipse is shown below with an example for an ellipse.



If you take the horizontal as your major axis, the distance **a** will correspond to the **Radius%** specified above. For the desired **b** use the following equation to find the value for **XAspect%**:

```
XAspect% = (32767& * b) / a
```

This equation could be done using long integer arithmetic at runtime, or done in the QuickBASIC interpreter, and the result would then be placed in your program. Note that **YAspect%** would be left as zero.

If you flip the ellipse so that the Y-axis is the major axis, then you would simply replace **YAspect%** into the above equation and leave **XAspect%** as zero.

■ **See Also:**

LineVE

ClsVE

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  The routine ClsVE clears the current EGA or VGA video screen.

- **Syntax:**

```
CALL ClsVE
```

Comments:

The following example will clear the VGA video screen after drawing a line.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'    'required for BYVAL's
SCREEN 12                  'sets the monitor in VGA mode

CALL LineVE (100, 100, 200, 200, 15)
WHILE INKEY$ = "": WEND

CALL ClsVE
```

DrawPointH

Assembler subroutine contained in GW.LIB

- **Purpose:**

  DrawPointH draws a pixel on a Hercules screen at the specified (X, Y) coordinate. This routine simulates the BASIC PSET routine, but does not require that MSHERC.COM be loaded.

- **Syntax:**

```
CALL DrawPointH(BYVAL XPos%, BYVAL YPos%, BYVAL PointColor%)
```

- **Where:**

  XPos% and **YPos%** specify an (X, Y) coordinate on the screen.

  PointColor% is a color value either 0 or 1.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following brief example shows how to plot a point at coordinate 100, 100 on a Hercules graphics screen without having to use the MSHERC.COM TSR program.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
CALL Screen3              'set the Hercules into graphics mode

CALL DrawPointH (100, 100, 1)
```

The DrawPointH routine can be used by the BASIC example routine which show the algorithm for lines and circles. These routines are LINEBAS.BAS and CIRCBAS.BAS. Later versions of Graphics Workshop will have more support for the Hercules screen mode.

- **See Also:**

  GetPointH%

DrawPointVE

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  DrawPointVE draws a pixel on an EGA or VGA screen at the specified (X, Y) coordinate. This routine simulates the BASIC PSET routine.

- **Syntax:**

```
CALL DrawPointVE(BYVAL XPos%, BYVAL YPos%, BYVAL PointColor%)
```

- **Where:**

  XPos% and **YPos%** specify an (X, Y) coordinate on the screen.

  PointColor% is a color value from 0 to 15.

- **Comments:**

  All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

  The following code fragment shows how to plot a pixel in the color blue on the VGA screen at location 300, 200.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in a VGA mode

CALL DrawPointVE (300, 200, 1)
```

  DrawPointVE is faster than QuickBASIC's PSET routine, as shown in the DEMOBNCH.BAS demonstration. DrawByteVE can set more than one pixel at a time, and GetPointVE% will return the color of any pixel.

- **See Also:**

  DrawByteVE, GetPointVE%

DrawPointVEOpts

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

DrawPointVEOpts draws a pixel on an EGA or VGA screen at the specified (X, Y) coordinate. This routine simulates the BASIC PSET routine with one benefit, the option of using a different logical operation than straight replacement. Like the LineVE and CircleVE routines, this routine can use a logical operation like OR, AND, or XOR when placing the pixel on the screen.

■ **Syntax:**

```
CALL DrawPointVEOpts (BYVAL XPos%, BYVAL YPos%, BYVAL PointColor%)
```

■ **Where:**

XPos% and **YPos%** specify an (X, Y) coordinate on the screen.

PointColor% is a color value from 0 to 15. If you add one of the values below to the color, then the associated logical operation will be performed.

VALUE	LOGICAL OPERATION
0	PSET (Replace)
2048	OR
4096	AND
6144	XOR

Any other values will produce unusual (and probably undesirable) effects.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The short example below shows how to use the XOR feature to invert the pixel at location 120, 300 of a VGA screen.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets monitor in a VGA mode

CALL DrawPointVEOpts (120, 300, 15 + 6144)
```

The DrawPointVEOpts routine is faster than the QuickBASIC PSET routine, as shown in the DEMOBNCH.BAS demonstration. The DrawByteVEOpts routine could be used to set more than one pixel at a

time. The GetPointVE% function will return the color value of any pixel on the screen.

■ **See Also:**

DrawByteVEOpts, GetPointVE%

GetPointH%

Assembler
function contained in GW.LIB

- **Purpose:**

  GetPointH% returns the color of the pixel at a specified (X, Y) coordinate.

- **Syntax:**

```
V% = GetPointH%(BYVAL XPos%, BYVAL YPos%)
```

- **Where:**

  XPos% and **YPos%** make up the (X, Y) coordinate.

  V% will receive a color value of either 0 or 1.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The code fragment below shows how to get the color of the pixel at location 100, 100 on the Hercules graphics screen.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
CALL Screen3              'sets Hercules into graphics mode

V% = GetPointH% (100, 100)
```

The above code fragment also demonstrates the Screen3 routine which sets the Hercules into graphics mode without the need for the TSR MSHERC.COM.

- **See Also:**

  DrawPointH

GetPointVE%

Assembler
function contained in GW.LIB

■ **Purpose:**

GetPointVE% returns the color of the pixel at a specified (X, Y) coordinate.

■ **Syntax:**

```
V% = GetPointVE%(BYVAL XPos%, BYVAL YPos%)
```

■ **Where:**

XPos% and **YPos%** make up the (X, Y) coordinate.

V% will receive a color value between 0 and 15 for the high-resolution EGA and VGA screen modes.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The next code fragment shows how to get the color at a specified location on the EGA high-resolution graphics screen.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in EGA mode

Colr% = GetPointVE% (200, 300)
```

There is also a routine for getting the color of a pixel which utilizes a special cache. This routine is called GetCacheVE% and is contained entirely in Appendix E.

■ **See Also:**

DrawPointVE, GetCacheVE%

GPrint0C2

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  GPrint0C2 prints a string on the 2-color CGA high-resolution screen in the specified color.

- **Syntax:**

```
CALL GPrint0C2 (BYVAL Row%, BYVAL Column%, Text$, BYVAL TextColor%)
```

- **Where:**

  Row% and **Column%** are the normal coordinates used by the BASIC LOCATE statement.

  Text$ is any text string.

  TextColor% holds the combined foreground and background colors. The following formula can be used to set the colors used:

```
TextColor% = Foreground% + (Background * 256)
```

Comments:

Many parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following example shows how to print a string to the CGA 2-color screen mode using this routine.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 2                  'sets the monitor in CGA mode

CALL GPrint0C2 (1, 10, "This text is on row 1", 1 + 0 * 256)
```

This routine is many times faster than the BASIC PRINT statement. It also allows you to specify a background color for that text string.

- **See Also:**

  GPrint0VE, GPrint0H

GPrint0H

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  GPrint0H prints a string on the 2-color high-resolution Hercules graphics screen.

- **Syntax:**

```
CALL GPrint0H (BYVAL Row%, BYVAL Column%, Text$, BYVAL TextColor%)
```

- **Where:**

  Row% and **Column%** are similar to the coordinates used by the BASIC LOCATE statement, but provide 90 columns instead of 80 in the Hercules high-resolution graphics screen.

  Text$ is any text string.

  TextColor% holds the combined foreground and background colors. The following formula can be used to set the colors used:

```
TextColor% = Foreground% + (Background * 256)
```

Comments:

Many parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The code fragment below shows how to print a string using inverse video with the Hercules graphics mode.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
CALL Screen3              'sets the Hercules in graphics mode

CALL GPrint0H (1, 10, "This Text is on row 1", 0 + 1 * 256)
```

This routine is many times faster than the BASIC PRINT statement. It also allows for you to specify a background color for that text string.

- **See Also:**

  GPrint0VE, GPrint0C2

GPrint0V256

Assembler subroutine contained in GW.LIB

■ **Purpose:**

GPrint0V256 prints a string on the 256-color VGA low-resolution graphics screen in a specified color.

■ **Syntax:**

```
CALL GPrint0V256 (BYVAL Row%, BYVAL Column%, Text$, BYVAL TextColor%)
```

■ **Where:**

Row% and **Column%** are the normal coordinates used by the BASIC LOCATE statement in SCREEN 13.

Text$ is any text string.

TextColor% holds the combined foreground and background colors. The following formula can be used to set the colors used:

```
TextColor% = Foreground% + (Background * 256)
```

Comments:

Many parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following example shows how to print a string to the VGA using the color blue for the foreground, and the color grey for the background.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 13                 'sets the monitor in VGA mode

GPrint0V256 1, 10, "This text is on row 1", 1 + 7 * 256
```

This routine is many times faster than the BASIC PRINT statement. It also allows you to specify a background color for that text string.

■ **See Also:**

GPrint0VE, GPrint1VE, GPrint2VE, GPrint3V256, GPrint3VE, GPrint0C2, GPrint0H

GPrint0VE

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  GPrint0VE prints a string on the 16-color EGA and VGA high-resolution graphics screens in a specified color.

- **Syntax:**

```
CALL GPrint0VE (BYVAL Row%, BYVAL Column%, Text$, BYVAL TextColor%)
```

- **Where:**

  Row% and **Column%** are the normal coordinates used by the BASIC LOCATE statement.

  Text$ is any text string.

  TextColor% holds the combined foreground and background colors. The following formula can be used to set the colors used:

```
TextColor% = Foreground% + (Background * 256)
```

Comments:

Many parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following example shows how to print a string to the VGA using the color blue for the foreground, and the color grey for the background.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'    'required for BYVAL's
SCREEN 12                  'sets the monitor in VGA mode

GPrint0VE 1, 10, "This text is on row 1", 1 + 7 * 256
```

This routine is many times faster than the BASIC PRINT statement. It also allows you to specify a background color for that text string.

- **See Also:**

  GPrint1VE, GPrint2VE, GPrint3VE, GPrint0C2, GPrint0H

GPrint1VE

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  GPrint1VE prints a string on the 16-color EGA and VGA high-resolution graphics screens in a specified color. This routine does not modify any of the background beneath the text.

- **Syntax:**

```
CALL GPrint1VE (BYVAL Row%, BYVAL Column%, Text$, BYVAL TextColor%)
```

- **Where:**

  Row% and **Column%** are the normal coordinates used by the BASIC LOCATE statement.

  Text$ is any text string.

  TextColor% holds the combined foreground and background colors. The following formula can be used to set the colors used:

```
TextColor% = Foreground% + (Background * 256)
```

  Although this is a standard formula for text colors with all of the GPrintXXX routines, in this case the background color is ignored and does not need to be placed in the above formula, but it will not matter if you leave it in the formula.

Comments:

Many parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The example below shows how to print a string to a VGA monitor in the color red that will write over any graphics objects, leaving the objects in the background.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

GPrint1VE 1, 10, "This text is on row 1", 4
```

This routine is many times faster than the BASIC PRINT statement. It also allows for the placement of text over even the most complex graphics images without altering them.

■ **See Also:**

GPrint0VE, GPrint2VE, GPrint3VE

GPrint2VE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

GPrint2VE prints a string on the 16-color EGA and VGA high-resolution graphics screens in a specified color. This routine allows text to be placed starting at any of the 350 lines on the EGA high-resolution display, or any of the 480 lines on the VGA display.

■ **Syntax:**

```
CALL GPrint2VE (BYVAL Line%, BYVAL Column%, Text$, BYVAL TextColor%)
```

■ **Where:**

Line% and **Column%** specify the starting location of the string using the mixed coordinate system where column values range from 1 to 80 and line values range from 0 to 479 on a VGA monitor. This is the difference between this routine and GPrint0VE.

Text$ is any text string.

TextColor% holds the combined foreground and background colors. The following formula can be used to set the colors used:

```
TextColor% = Foreground% + (Background * 256)
```

Comments:

Parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The example below shows how to print a string of text, with a foreground color of white and a background color of red, at any Y coordinate on the VGA screen.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

GPrint2VE 103, 10, "This text is on line 103", 15 + 4 * 256
```

This routine is many times faster than the BASIC PRINT statement. It also allows you to specify a background color for that text string.

■ **See Also:**

GPrint0VE, GPrint1VE, GPrint3VE

GPrint3V256

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

GPrint3V256 prints a string on the 256-color VGA low-resolution graphics screen in a specified color. This routine allows text to be placed starting at any of the 200 lines on the VGA low-resolution display. This routine does not modify any of the background beneath the text.

■ **Syntax:**

```
CALL GPrint3V256 (BYVAL Line%, BYVAL Column%, Text$, BYVAL TextColor%)
```

■ **Where:**

Line% and **Column%** specify the starting location of the string using the mixed coordinate system where column values range from 1 to 80 and line values range from 0 to 199.

Text$ is any text string.

TextColor% holds the combined foreground and background colors. The following formula can be used to set the colors used:

```
TextColor% = Foreground% + (Background * 256)
```

Although this is a standard formula for text colors with all of the GPrintXXX routines, in this case the background color is ignored and does not need to be placed in the above formula, but it will not matter if you leave it in the formula.

Comments:

Many parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The example below shows how to print a string of text over any objects on the screen leaving the objects in the background. The text can be printed at any one of the VGA's 200 line positions.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'    'required for BYVAL's
SCREEN 13                  'sets the monitor in VGA mode

GPrint3V256 102, 2, "This text is on line 102", 14
```

This routine is many times faster than the BASIC PRINT statement. It also allows for the placement of text over even the most complex graphics images without altering them.

■ **See Also:**

GPrint0V256, GPrint0VE, GPrint1VE, GPrint2VE

GPrint3VE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

GPrint3VE prints a string on the 16-color EGA and VGA high-resolution graphics screens in a specified color. This routine allows text to be placed starting at any of the 350 lines on the EGA high-resolution display, or any of the 480 lines on the VGA display. This routine does not modify any of the background beneath the text.

■ **Syntax:**

```
CALL GPrint3VE (BYVAL Line%, BYVAL Column%, Text$, BYVAL TextColor%)
```

■ **Where:**

Line% and **Column%** specify the starting location of the string using the mixed coordinate system where column values range from 1 to 80 and line values range from 0 to 479 on a VGA monitor. This is the difference between this routine and GPrint1VE.

Text$ is any text string.

TextColor% holds the combined foreground and background colors. The following formula can be used to set the colors used:

```
TextColor% = Foreground% + (Background * 256)
```

Although this is a standard formula for text colors with all of the GPrintXXX routines, in this case the background color is ignored and does not need to be placed in the above formula, but it will not matter if you leave it in the formula.

Comments:

Many parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The example below shows how to print a string of text over any objects on the screen leaving the objects in the background. The text can be printed at any one of the VGA's 480 line positions.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

GPrint3VE 302, 2, "This text is on line 302", 14
```

This routine is many times faster than the BASIC PRINT statement. It also allows for the placement of text over even the most complex graphics images without altering them.

■ **See Also:**

GPrint0VE, GPrint1VE, GPrint2VE

LineBF2VE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

LineBF2VE draws a filled box on an EGA or VGA high-resolution screen. Unlike LineBFVE this routine cannot utilize OR, AND, XOR, and PSET operations for drawing the filled box. This routine is however 40% faster than LineBFVE.

■ **Syntax:**

```
CALL LineBF2VE (BYVAL x1%, BYVAL y1%, BYVAL x2%, BYVAL y2%,_
  BYVAL LineColor%)
```

■ **Where:**

The coordinate pairs (**x1%**, **y1%**) and (**x2%**, **y2%**) are within the range of the screen.

LineColor% is the color of the line.

To create a filled box in the color grey, you would pass LineBF2VE the color value shown in the following example.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in an EGA mode

LineColor% = 7
CALL LineBF2VE (100, 100, 200, 200, LineColor%)
```

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

■ **See Also:**

LineVE, LineBFVE, LineBVE

LineBFVE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

LineBFVE draws a filled box on an EGA or VGA high-resolution screen. This routine can utilize OR, AND, XOR, and PSET operations for drawing the filled box.

■ **Syntax:**

```
CALL LineBFVE (BYVAL x1%, BYVAL y1%, BYVAL x2%, BYVAL y2%,_
  BYVAL LineColor%)
```

■ **Where:**

The coordinate pairs (**x1%**, **y1%**) and (**x2%**, **y2%**) are within the range of the screen.

LineColor% is the color of the line. Add one of the values below in the table to change the logical operation performed when drawing the filled box.

VALUE	LOGICAL OPERATION
0	PSET (Replace)
2048	OR
4096	AND
6144	XOR

Any other values will produce unusual (and probably undesirable) effects.

To create a rubber-banding effect with the color grey, you would pass LineBFVE the color value shown in the following example.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in an EGA mode

LineColor% = 7 + 6144
CALL LineBFVE (100, 100, 200, 200, LineColor%)
```

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

■ **See Also:**

LineVE, LineBF2VE, LineBVE

LineBVE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

LineBVE draws a box outline on an EGA or VGA high-resolution screen. This routine can utilize OR, AND, XOR, and PSET operations for drawing the box.

■ **Syntax:**

```
CALL LineBVE (BYVAL x1%, BYVAL y1%, BYVAL x2%, BYVAL y2%, _
  BYVAL LineColor%)
```

■ **Where:**

The coordinate pairs (**x1%, y1%**) and (**x2%, y2%**) are within the range of the screen.

LineColor% is the color of the line. Add one of the values below in the table to change the logical operation performed when drawing the line.

VALUE	LOGICAL OPERATION
0	PSET (Replace)
2048	OR
4096	AND
6144	XOR

Any other values will produce unusual (and probably undesirable) effects.

To create a rubber-banding effect with the color grey, you would pass LineBVE the color value shown in the following example.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in an EGA mode

LineColor% = 7 + 6144
CALL LineBVE (100, 100, 200, 200, LineColor%)
```

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

■ **See Also:**

LineVE, LineBFVE

LineStepVE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

LineStepVE draws a line on an EGA or VGA high-resolution screen using step values from the last point drawn to. This routine can utilize OR, AND, XOR, and PSET operations for drawing the line.

■ **Syntax:**

```
CALL LineStepVE (BYVAL StepX1%, BYVAL StepY1%, BYVAL StepX2%, BYVAL_
  StepY2%, BYVAL LineColor%)
```

■ **Where:**

The coordinate pairs (**StepX1%, StepY1%**) and (**StepX2%, StepY2%**) are within the range of the screen and are distances from the last point drawn to.

LineColor% is the color of the line. Add one of the values below in the table to change the logical operation performed when drawing the line.

VALUE	LOGICAL OPERATION
0	PSET (Replace)
2048	OR
4096	AND
6144	XOR

Any other values will produce unusual (and probably undesirable) effects.

To create a rubber-banding effect with the color grey, you would pass LineStepVE the color value shown in the following example. Note that the line will be drawn from coordinate (100, 100) to (200, 200).

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in an EGA mode

LineColor% = 7 + 614
CALL SetLastCoord(50, 50)
CALL LineStepVE (50, 50, 150, 150, LineColor%)
```

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

- **See Also:**

  LineVE, LineToStepVE

LineToVE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

LineToVE draws a line on an EGA or VGA high-resolution screen from the last coordinate drawn to, to the coordinate specified to this routine. This routine can utilize OR, AND, XOR, and PSET operations for drawing the line.

■ **Syntax:**

```
CALL LineToVE (BYVAL ToX%, BYVAL ToY%, BYVAL LineColor%)
```

■ **Where:**

The coordinate pair (**ToX%, ToY%**) is within the range of the screen.

LineColor% is the color of the line. Add one of the values below in the table to change the logical operation performed when drawing the line.

VALUE	LOGICAL OPERATION
0	PSET (Replace)
2048	OR
4096	AND
6144	XOR

Any other values will produce unusual (and probably undesirable) effects.

To create a rubber-banding effect with the color grey, you would pass LineToVE the color value shown in the following example.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in an EGA mode

LineColor% = 7 + 6144
CALL SetLastCoord(100, 100)
CALL LineToVE (200, 200, LineColor%)
```

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

■ **See Also:**

LineVE, LineToStepVE

LineToStepVE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

LineToStepVE draws a line on an EGA or VGA high-resolution screen from the last coordinate drawn to, to the point a specified distance away. This routine can utilize OR, AND, XOR, and PSET operations for drawing the line.

■ **Syntax:**

```
CALL LineToStepVE (BYVAL StepX%, BYVAL StepY%, BYVAL LineColor%)
```

■ **Where:**

The coordinate pair **(StepX%, StepY%)** is within the range of the screen and specifies a distance from the last point drawn to.

LineColor% is the color of the line. Add one of the values below in the table to change the logical operation performed when drawing the line.

VALUE	LOGICAL OPERATION
0	PSET (Replace)
2048	OR
4096	AND
6144	XOR

Any other values will produce unusual (and probably undesirable) effects.

To create a rubber-banding effect with the color grey, you would pass LineToStepVE the color value shown in the following example.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in an EGA mode

LineColor% = 7 + 6144
CALL SetLastCoord (100, 100)
CALL LineToStepVE (100, 100, LineColor%)
```

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

See Also:LineVE, LineStepVE, LineToStepVE

LineVE

Assembler
subroutine contained in GW.LIB

Purpose:

LineVE draws a line on an EGA or VGA high-resolution screen. This routine can utilize OR, AND, XOR, and PSET operations for drawing the line.

Syntax:

```
CALL LineVE (BYVAL x1%, BYVAL y1%, BYVAL x2%, BYVAL y2%, _
  BYVAL LineColor%)
```

Where:

The coordinate pairs (**x1%, y1%**) and (**x2%, y2%**) are within the range of the screen.

LineColor% is the color of the line. Add one of the values below in the table to change the logical operation performed when drawing the line.

VALUE	LOGICAL OPERATION
0	PSET (Replace)
2048	OR
4096	AND
6144	XOR

Any other values will produce unusual (and probably undesirable) effects.

To create a rubber-banding effect with the color grey, you would pass LineVE the color value shown in the following example.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in an EGA mode

LineColor% = 7 + 6144
CALL LineVE (100, 100, 200, 200, LineColor%)
```

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

See Also:

CircleVE, DrawPointVE, LineBFVE, LineBVE, LineStepVE, LineToVE, LineToStepVE

Screen0

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

Screen0 sets a monitor into text mode. This routine should be used at the end of your programs if you have used one of the other screen setting routines in Graphics Workshop.

■ **Syntax:**

```
CALL Screen0(KeepData%)
```

■ **Where:**

KeepData% is a true or false value which tells the routine whether or not to clear the screen when setting the screen. This gives you an ability BASIC does not have. If you set this value to anything other than 0, all the information on the screen will remain in memory and you can even return to it. A value of 0 will clear the screen at the same time the video mode is set.

Comments:

The following example sets the monitor into text mode after using another of the Graphics Workshop Screen routines which had previously set the monitor into the VGA graphics mode.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
CALL Screen12(0)          'sets the monitor in VGA mode

CALL Screen0(0)           'sets the monitor into text mode
```

If you're going to utilize the KeepData feature of this routine, you will need to understand a little about how the video memory is structured for this screen mode. Most programmers will pass only a value of 0 which makes this work the same as the BASIC SCREEN statement equivalent. To take advantage of the KeepData feature, examine the example program DEMOSCRN.BAS.

■ **See Also:**

Screen1, Screen2, Screen3, Screen7, Screen9, Screen11, Screen12, Screen13

Screen1

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

Screen1 sets an CGA monitor into the 320 by 200 pixel 4-color mode. This is a replacement for the SCREEN 1 statement in BASIC.

■ **Syntax:**

```
CALL Screen1(KeepData%)
```

■ **Where:**

KeepData% is a true or false value which tells the routine whether or not to clear the screen when setting the screen. This gives you an ability BASIC does not have. If you set this value to anything other than 0, all the information on the screen will remain in memory and you can even return to it. A value of 0 will clear the screen at the same time the video mode is set.

Comments:

The following example sets the monitor into the CGA 4-color graphics mode, and clears the screen at the same time.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's

CALL Screen1(0)           'sets the monitor in CGA mode
```

If you're going to utilize the KeepData feature of this routine, you will need to understand a little about how the video memory is structured for this screen mode. Most programmers will pass only a value of 0 which makes this work the same as the BASIC SCREEN statement equivalent. To take advantage of the KeepData feature, examine the example program DEMOSCRN.BAS.

To return to text mode at the completion of your program, use the routine Screen0. Otherwise your program will return to DOS while still in a graphics mode.

■ **See Also:**

Screen0, other screen setting routines

Screen2

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

Screen2 sets an CGA monitor into the 640 by 200 pixel 2-color mode. This is a replacement for the SCREEN 2 statement in BASIC.

■ **Syntax:**

```
CALL Screen2(KeepData%)
```

■ **Where:**

KeepData% is a true or false value which tells the routine whether or not to clear the screen when setting the screen. This gives you an ability BASIC does not have. If you set this value to anything other than 0, all the information on the screen will remain in memory and you can even return to it. A value of 0 will clear the screen at the same time the video mode is set.

Comments:

The following example sets the monitor into the CGA 2-color graphics mode, and clears the screen at the same time.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's

CALL Screen2(0)           'sets the monitor in CGA mode
```

If you're going to utilize the KeepData feature of this routine, you will need to understand a little about how the video memory is structured for this screen mode. Most programmers will pass only a value of 0 which makes this work the same as the BASIC SCREEN statement equivalent. To take advantage of the KeepData feature, examine the example program DEMOSCRN.BAS.

To return to text mode at the completion of your program, use the routine Screen0. Otherwise your program will return to DOS while still in a graphics mode.

■ **See Also:**

Screen0, other screen setting routines

Screen3

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

Screen3 sets a Hercules monitor into the 720 by 350 pixel 2-color mode. This is a replacement for the SCREEN 3 statement in BASIC.

■ **Syntax:**

```
CALL Screen3
```

Comments:

If you use this routine to set the screen mode you will NOT need to run MSHERC.COM for simple Hercules support. If you are using the PCXArrayH or DispPCXH routines to display a PCX file, or can achieve all of your Hercules graphics needs using the BASIC POKE statement, then this routine will serve your Hercules graphics needs without requiring MSHERC.COM to be loaded.

The following example shows how to turn on the Hercules graphics mode without the need of the MSHERC.COM Terminate and Stay Resident utility. Specifying not to keep the data will tell the routine to clear the screen at the same time.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's

CALL Screen3(0)           'sets the Hercules into graphics mode
```

If you need more of the features BASIC provides using the TSR MSHERC.COM, take a look at the Graphics Workshop function HercThere% which determines whether or not the TSR is loaded in memory.

To return to text mode at the completion of your program, use the routine Screen0. Otherwise your program will return to DOS while still in a graphics mode.

■ **See Also:**

HercThere%, MSHERC.COM, Screen0

Screen7

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

Screen7 sets an EGA monitor into the 640 by 350 pixel 2-color mode. This is a replacement for the SCREEN 7 statement in BASIC.

■ **Syntax:**

```
CALL Screen7(KeepData%)
```

■ **Where:**

KeepData% is a true or false value which tells the routine whether or not to clear the screen when setting the screen. This gives you an ability BASIC does not have. If you set this value to anything other than 0, all the information on the screen will remain in memory and you can even return to it. A value of 0 will clear the screen at the same time the video mode is set.

Comments:

The following example sets the EGA monitor into a monochrome graphics mode, and clears the screen.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
CALL Screen7(0)           'sets the monitor in EGA mono
```

If you're going to utilize the KeepData feature of this routine, you will need to understand a little about how the video memory is structured for this screen mode. Most programmers will pass only a value of 0 which makes this work the same as the BASIC SCREEN statement equivalent. To take advantage of the KeepData feature, examine the example program DEMOSCRN.BAS.

To return to text mode at the completion of your program, use the routine Screen0. Otherwise your program will return to DOS while still in a graphics mode.

■ **See Also:**

Screen0, other screen setting routines

Screen9

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

Screen9 sets an EGA monitor into the 640 by 350 pixel 16-color mode. This is a replacement for the SCREEN 9 statement in BASIC.

■ **Syntax:**

```
CALL Screen9(KeepData%)
```

■ **Where:**

KeepData% is a true or false value which tells the routine whether or not to clear the screen when setting the screen. This gives you an ability BASIC does not have. If you set this value to anything other than 0, all the information on the screen will remain in memory and you can even return to it. A value of 0 will clear the screen at the same time the video mode is set.

Comments:

The example below shows how to set the EGA into its high-resolution screen mode, and clear the screen at the same time.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's

CALL Screen9(0)           'sets the monitor in EGA mode
```

If you're going to utilize the KeepData feature of this routine, you will need to understand a little about how the video memory is structured for this screen mode. Most programmers will pass only a value of 0 which makes this work the same as the BASIC SCREEN statement equivalent. To take advantage of the KeepData feature, examine the example program DEMOSCRN.BAS.

To return to text mode at the completion of your program, use the routine Screen0. Otherwise your program will return to DOS while still in a graphics mode.

■ **See Also:**

Screen0, other screen setting routines

Screen11

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

Screen11 sets a VGA monitor into the 640 by 480 pixel 2-color mode. This is a replacement for the SCREEN 11 statement in BASIC.

■ **Syntax:**

```
CALL Screen11(KeepData%)
```

■ **Where:**

KeepData% is a true or false value which tells the routine whether or not to clear the screen when setting the screen. This gives you an ability BASIC does not have. If you set this value to anything other than 0, all the information on the screen will remain in memory and you can even return to it. A value of 0 will clear the screen at the same time the video mode is set.

Comments:

The brief code fragment below shows how to set the VGA monitor into a high-resolution monochrome mode. The call to Screen11 will clear the screen at the same time.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's

CALL Screen11(0)          'sets the monitor in VGA mono
```

If you're going to utilize the KeepData feature of this routine, you will need to understand a little about how the video memory is structured for this screen mode. Most programmers will pass only a value of 0 which makes this work the same as the BASIC SCREEN statement equivalent. To take advantage of the KeepData feature, examine the example program DEMOSCRN.BAS.

To return to text mode at the completion of your program, use the routine Screen0. Otherwise your program will return to DOS while still in a graphics mode.

■ **See Also:**

Screen0, other screen setting routines

Screen12

Assembler subroutine contained in GW.LIB

■ **Purpose:**

Screen12 sets a VGA monitor into the 640 by 480 pixel 16-color mode. This is a replacement for the SCREEN 12 statement in BASIC.

■ **Syntax:**

```
CALL Screen12(KeepData%)
```

■ **Where:**

KeepData% is a true or false value which tells the routine whether or not to clear the screen when setting the screen. This gives you an ability BASIC does not have. If you set this value to anything other than 0, all the information on the screen will remain in memory and you can even return to it. A value of 0 will clear the screen at the same time the video mode is set.

Comments:

The following example sets the VGA into its high-resolution color mode, and clears the screen at the same time.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's

CALL Screen12(0)          'sets the monitor in VGA mode
```

If you're going to utilize the KeepData feature of this routine, you will need to understand a little about how the video memory is structured for this screen mode. Most programmers will pass only a value of 0 which makes this work the same as the BASIC SCREEN statement equivalent. To take advantage of the KeepData feature, examine the example program DEMOSCRN.BAS.

To return to text mode at the completion of your program, use the routine Screen0. Otherwise your program will return to DOS while still in a graphics mode.

■ **See Also:**

Screen0, other screen setting routines

Screen13

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

Screen13 sets a VGA monitor into the 320 by 200 pixel 256-color mode. This is a replacement for the SCREEN 13 statement in BASIC.

■ **Syntax:**

```
CALL Screen13(KeepData%)
```

■ **Where:**

KeepData% is a true or false value which tells the routine whether or not to clear the screen when setting the screen. This gives you an ability BASIC does not have. If you set this value to anything other than 0, all the information on the screen will remain in memory and you can even return to it. A value of 0 will clear the screen at the same time the video mode is set.

Comments:

The following example sets the VGA monitor into its 256-color mode, and clears the screen at the same time.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's

CALL Screen13(0)          'sets the monitor in 256-color mode
```

If you're going to utilize the KeepData feature of this routine, you will need to understand a little about how the video memory is structured for this screen mode. Most programmers will pass only a value of 0 which makes this work the same as the BASIC SCREEN statement equivalent. To take advantage of the KeepData feature, examine the example program DEMOSCRN.BAS.

To return to text mode at the completion of your program, use the routine Screen0. Otherwise your program will return to DOS while still in a graphics mode.

■ **See Also:**

Screen0, other screen setting routines

Section 3:
General Screen Manipulation Routines

The routines in this section are routines for which there is no BASIC equivalent. Some of the routines move regions of the graphics screen to other locations. The GScrollVE routine will allow any region of a graphics screen to be scrolled in any direction.

ClearScreenArray

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

The routine ClearScreenArray is used to reset the screen array maintained by the various GPrint??? routines.

■ **Syntax:**

```
CALL ClearScreenArray
```

Comment:

The example below resets the screen array and then checks the character value at position 1,1.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'                    'required for BYVAL's
SCREEN 12                                  'sets the monitor in VGA mode

CALL ClearScreenArray

Char% = GetCharacter%(1, 1)
'Char% will equal 32 (a space) at this point.
```

■ **See Also:**

GetCharacter%

ClearVE

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  The routine ClearVE will reset the EGA and VGA registers to a known state.

- **Syntax:**

```
CALL ClearVE
```

Comment:

The example below clears the VGA registers.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

CALL ClearVE
```

There may be no need for this routine unless you modify the EGA or VGA registers on your own. If you do modify the EGA or VGA registers, this routine can be used to reset all of them to a known state.

DrawByteVE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

DrawByteVE will draw a byte to an EGA or VGA screen at the specified physical location in the specified color. Only those bits specified in the bit pattern will be modified. Using this routine allows you to plot up to 8 pixels at a time.

■ **Syntax:**

```
CALL DrawByteVE(BYVAL ScreenPosition%, BYVAL BitPattern%, _
  BYVAL ByteColor%)
```

■ **Where:**

ScreenPosition% is an actual memory address for the EGA screen. The function MakeAddressVE% can provide you with the EGA memory address if you need it.

BitPattern% is a byte value for which every bit set in its binary equivalent will be colored on the screen.

ByteColor% is a color from 0 to 15 for the bits to be set. Only those bits specified by the BitPattern% parameter will receive the color in Byte-Color%.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following example sets the first eight horizontal bits on the screen to the color red.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

Address% = 0              'first address of the screen
BitPattern% = 255         'set all bits

CALL DrawByteVE (Address%, BitPattern%, 4)
```

The DrawByteVE routine provides the second fastest drawing speed to the screen. The fastest drawing speed can be obtained by using the LineVE statement.

■ **See Also:**

DrawPointVE, LineVE, MakeAddressVE%

DrawByteVEOpts

Assembler subroutine contained in GW.LIB

■ **Purpose:**

DrawByteVEOpts will draw a byte to an EGA or VGA screen at the specified physical location in the specified color. Only those bits specified in the bit pattern will be modified. Using this routine allows you to plot up to 8 pixels at a time. In addition, this routine can use any of the logical operators OR, AND, or XOR.

■ **Syntax:**

```
CALL DrawByteVEOpts(BYVAL ScreenPosition%, BYVAL BitPattern%, _
  BYVAL ByteColor%)
```

■ **Where:**

ScreenPosition% is an actual memory address for the EGA screen. The function MakeAddressVE% can provide you with the EGA memory address if you need it.

BitPattern% is a byte value for which every bit set in its binary equivalent will be colored on the screen.

ByteColor% is a color from 0 to 15 for the bits to be set. Only those bits specified by the BitPattern% parameter will receive the color in Byte-Color%. If you add one of the below values to the color, then the associated logical operator will be performed.

VALUE	LOGICAL OPERATION
0	PSET (Replace)
2048	OR
4096	AND
6144	XOR

Any other values will produce unusual (and probably undesirable) effects.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following example uses to logical operation OR, to combine the color blue with the first eight horizontal pixels on the screen.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'    'required for BYVAL's
SCREEN 12                  'sets the monitor in VGA mode

Address% = 0               'first address on the screen
BitPattern% = 255          'set all bits
CALL DrawByteVEOpts (Address%, BitPattern%, 1 + 2048)
```

The DrawByteVEOpts routine provides the second fastest drawing speed to the screen. The fastest drawing speed can be obtained by using the LineVE statement.

- **See Also:**

  DrawByteVE, DrawPointVE, LineVE, MakeAddressVE%

Fade2EGA

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

Fade2EGA uses a random sequence of points to transfer images from one video page to another. This routine is meant for use with EGA adapters.

■ **Syntax:**

```
CALL Fade2EGA(BYVAL GridX%, BYVAL GridY%, BYVAL RandomPortion%)
```

■ **Where:**

The video screen is broken up into a grid where each region is 40 pixels wide by 8 pixels high. **GridX%** is a number between 1 and 16. **GridY%** will range between 1 and 44.

The variable **RandomPortion%** is a number between 1 and 5. It refers to one of five pre-configured random patterns for the grid region. Placing all five patterns on one grid region will create a solid image on that grid region.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The below example draws some images, copies them to the second video page, and then uses the Fade2EGA routine on the upper-left grid region to transfer the image.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in EGA mode

'some sample graphics
LINE (0, 0) - (639, 349), 1, B
LINE (5, 5) - (100, 40), 2, BF
CIRCLE (30, 30), 27, 4
PAINT (30, 30), 4, 4
PCOPY 0, 1                'copy to second video page

CLS                       'clear the screen to see the effect
FOR T = 1 TO 5
  CALL Fade2EGA (1, 1, T)
NEXT
```

The sequence is pseudo random and is specified by the data in FADEDAT2.GW.

When a grid region is specified and a RandomPortion% is specified, it will transfer one-fifth of the screen image. Each grid region must be specified a total of five times, each time with a different random pattern specified to transfer the entire image to that region. The order for the random patterns is not important, as long as every region receives each RandomPortion at least once.

It is advisable to use this routine and this effect when the overall background of the two screen images is going to be similar. Any time when there is a large portion of the screen which will not be changed by this routine, the effect is much more pleasing to the eye and doesn't appear rigid.

The BASIC subroutines NightFall and RandomFade use this routine to transfer a screen from the background, and should be utilized as an example for this routine.

■ **See Also:**

FadeEGA, NightFall, RandomFade

FadeEGA

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

FadeEGA uses a random sequence of points to transfer images from one video page to another. This routine is meant for use with EGA adapters.

■ **Syntax:**

```
CALL FadeEGA(BYVAL GridX%, BYVAL GridY%, BYVAL RandomPortion%)
```

■ **Where:**

The video screen is broken up into a grid where each region is 160 pixels wide by 32 pixels high. **GridX%** is a number between 1 and 4. GridY% will range between 1 and 11.

The variable **RandomPortion%** is a number between 1 and 10. It refers to one of ten pre-configured random patterns for the grid region. Placing all ten patterns on one grid region will create a solid image on that grid region.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The below example draws some images, copies them to the second video page, and then uses the FadeEGA routine on the upper-left grid region to transfer the image.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in EGA mode

'some sample graphics
LINE (0, 0) - (639, 349), 1, B
LINE (10, 10) - (100, 40), 2, BF
CIRCLE (30, 30), 25, 4
PAINT (30, 30), 4, 4
PCOPY 0, 1                'copy to second video page
CLS                       'clear the screen to see the effect
FOR T = 1 TO 10
  CALL FadeEGA (1, 1, T)
NEXT
```

The sequence is pseudo random and is specified by the data in FADEDATA.GW.

When a grid region is specified and a RandomPortion% is specified, it will transfer one-tenth of the screen image. Each grid region must be specified a total of ten times, each time with a different random portion specified to transfer the entire image to that region. The order for the random portions is not important, as long as every region receives each RandomPortion at least once.

It is advisable to use this routine and this effect when the overall background of the two screen images is going to be similar. Any time when there is a large portion of the screen which will not be changed by this routine, the effect is much more pleasing to the eye and doesn't appear rigid.

The BASIC subroutines NightFall and RandomFade use this routine to transfer a screen from the background, and should be utilized as an example for this routine.

■ **See Also:**

Fade2EGA, NightFall, RandomFade

GetCharacter%

Assembler
function contained in GW.LIB

- **Purpose:**

  The function GetCharacter% returns the ASCII character value stored at the specified screen position.

- **Syntax:**

```
ASCII% = GetCharacter%(BYVAL Row%, BYVAL Col%)
```

- **Where:**

  Row% and **Col%** represent the location on the screen to inquire about.

- **Comments:**

  All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

  The following example gets the character value at location 1,1.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

CALL GPrintOVE (1, 1, "Hello", 15)

Char% = GetCharacter%(1, 1)
```

  Note that this routine doesn't read video memory to determine the character value on the screen. To save the time that would be required to interpret pixel patterns, the Graphics Workshop Print routines put the character values into a screen array 80 characters wide by 25 lines high.

- **See Also:**

  ClearScreenArray

GetLastXCoord%

Assembler
function contained in GW.LIB

■ **Purpose:**

GetLastXCoord% returns the value of the last x coordinate drawn to.

■ **Syntax:**

```
X% = GetLastXCoord%
```

Comment:

The following example demonstrates how to obtain the x coordinate value of the drawing cursor's location after a Graphics Workshop primitive has been used.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

CALL LineStep(103, 56, 15)

X% = GetLastXCoord%
```

■ **See Also:**

GetLastYCoord%, SetLastCoord

GetLastYCoord%

Assembler
function contained in GW.LIB

■ **Purpose:**

GetLastYCoord% returns the value of the last y coordinate drawn to.

■ **Syntax:**

```
Y% = GetLastYCoord%
```

Comment:

The following example demonstrates how to obtain the y coordinate value of the drawing cursor's location after a Graphics Workshop primitive has been used.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

CALL LineStep(103, 56, 15)

Y% = GetLastYCoord%
```

■ **See Also:**

GetLastXCoord%, SetLastCoord

GetScreenMode%

Assembler function contained in GW.LIB

- **Purpose:**

  GetScreenMode% returns the BIOS video mode currently in use.

- **Syntax:**

```
BIOSMode% = GetScreenMode%
```

Comments:

The following example checks to see if the BIOS video mode is greater than 13, meaning it is an EGA or VGA video mode.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'    'required for BYVAL's
SCREEN 12                  'sets the monitor in VGA mode

BIOSMode% = GetScreenMode%

IF BIOSMode%  13 THEN PRINT "EGA or VGA"
```

GMove1VE

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  GMove1VE moves any rectangular block on any video page to any location on any video page.

- **Syntax:**

```
CALL GMove1VE (BYVAL FromCol%, BYVAL FromLine%, BYVAL
  Cols%, BYVAL Lines%, BYVAL DestCol%, BYVAL DestLine%)
```

- **Where:**

  FromCol% and **FromLine%** specify the upper corner of the region to be moved. **Cols%** and **Lines%** specify the size of the region to be moved. These coordinates follow a mixed coordinate window system where column values range from 1 to 80 and line values range from 0 to 349 on an EGA monitor.

  DestCol% and **DestLine%** specify a mixed coordinate value which tells the upper-left corner where the image will be placed.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following code fragment copies an image from the right half of the page to the left half of the page.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

CALL SetGWPages(0, 0)     'make destination the same as the source

CALL GMove1VE(41, 0, 40, 480, 1, 0)
```

The default pages for this routine and others are page 1 for the source information and page 0 for the destination of the image. The source and destination pages can be re-directed using the SetGWPages routine.

When using the GMove1VE routine to move an image across the screen, you must take into account that the image is being overwritten while it is being moved. For these uses, look at the use of the GMove2VE routine in DEMOMOVE.BAS and the GScrollVE routine in DEMOSCRL.BAS to move images flawlessly.

■ **See Also:**

GMove2VE, GMove3VE, TransferEGA, SetGWPages

GMove2VE

Assembler
subroutine contained in GW.LIB

Purpose:

GMove2VE will save and restore any rectangular region of the screen to a video memory location which you specify. This routine uses screen memory to store the image. This approach has two advantages: Graphics saves and restores require one-fourth the instructions of other save and restore routines, and graphics memory is often not used and is therefore less costly to the programmer than using general memory.

Syntax:

```
DestSegment% = &HA800
CALL GMove2VE (BYVAL FromCol%, BYVAL FromLine%, BYVAL Cols%, BYVAL_
  Lines%, BYVAL DestSegment%, BYVAL Direction%)
```

Where:

FromCol% and **FromLine%** specify the upper corner of the region to be moved. **Cols%** and **Lines%** specify the size of the region to be moved. These coordinates follow a mixed coordinate window system where column values range from 1 to 80 and line values range from 0 to 479 on a VGA monitor.

The variable **DestSegment%** provides the routine with a location to send the information. This segment value should be within the range of EGA or VGA graphics memory available.

The variable **Direction%** decides whether the image will be saved or restored. A zero saves the image. Any other value will restore the image.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following example saves and restores the upper-left corner 10 column by 100 lines region of the screen.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

LINE (0, 0) - (79, 99), 1, B
'save the image
CALL GMove2VE (1, 0, 10, 100, &HAA00, 0)

CLS
'restore the image
CALL GMove2VE (1, 0, 10, 100, &HAA00, -1)
```

The default page for this routine is page 0. Depending upon the value in the parameter Direction%, the default page refers to the destination or the source for the image. The default page can be re-directed using the routine SetGWPages.

The beginning of the EGA's high-resolution second screen starts at &HA800. On the EGA display there is 128K free for the storage of images.

The VGA high-resolution mode doesn't have a second screen per se, but we still can use this routine. For the VGA the first unused graphics segment would be &HAA00 as shown in the above code example. The VGA has only 96K of available memory for the storage of images.

The following formula will help to calculate the amount of memory used by an image saved with this routine:

```
MemUsed% = Cols% * Lines% * 4
```

To use this to determine the next segment where graphics images can be stored, use

```
NextSegment% = ThisSegment% + MemUsed% \ 64 + 1
```

where ThisSegment% is the segment where the current graphics image is being stored.

This routine is a vital part of saving graphics images for use in the graphics PullDownG menus and in the VertMenuG routine.

■ **See Also:**

GMove1VE, GMove3VE, GMove4VE, SetGWPages

GMove3VE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

GMove3VE moves any rectangular block on any video page to any location on any video page. This routine has the ability to create an interlacing effect by skipping a specified number of lines between lines transferred.

■ **Syntax:**

```
CALL GMove3VE (BYVAL FromCol%, BYVAL FromLine%, BYVAL Cols%, BYVAL _
   Lines%, BYVAL DestCol%, BYVAL DestLine%, BYVAL SkipLines%)
```

■ **Where:**

FromCol% and **FromLine%** specify the upper corner of the region to be moved. **Cols%** and **Lines%** specify the size of the region to be moved. These coordinates follow a mixed coordinate window system where column values range from 1 to 80 and line values range from 0 to 349 on an EGA monitor.

DestCol% and **DestLine%** specify a mixed coordinate value which tells the upper-left corner where the image will be placed.

SkipLines% tells the routine how many lines to skip for every line copied within the region.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The SkipLines parameter is usually set to 1 which tells the routine to skip every other line. Also, it is common for this routine to be called once to bring in half of the lines and once more to bring in the other half. The BASIC routine SplitHorizontal makes use of this routine to bring in a graphics image. The demo DEMOGW.BAS makes use of this routine to display the opening title page.

The following example shows how to transfer an interlaced image of the right half of the screen to the left half of the screen.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

SkipLines% = 1
CALL GMove3VE (41, 0, 40, 480, 1, 0, SkipLines%)
```

This routine works exactly like GMove1VE if you specify SkipLines% = 0.

The default pages for this routine are page 1 for the source information and page 0 for the destination of the image. The source and destination pages can be re-directed using the routine SetGWPages.

■ **See Also:**

GMove1VE, GMove2VE, SetGWPages, SplitHorizontal, TransferEGA

GMove4VE

Assembler
subroutine contained in GW.LIB

Purpose:

GMove4VE will save and restore any rectangular region of the screen to an array you specify.

Syntax:

```
CALL GMove4VE (BYVAL FromCol%, BYVAL FromLine%, BYVAL Cols%, BYVAL _
   Lines%, BYVAL DestSegment%, BYVAL Direction%)
```

Where:

FromCol% and **FromLine%** specify the upper corner of the region to be moved. **Cols%** and **Lines%** specify the size of the region to be moved. These coordinates follow a mixed coordinate window system where column values range from 1 to 80 and line values range from 0 to 479 on a VGA monitor.

The variable **DestSegment%** provides the routine with a location to send the information. This segment value is determined by finding the segment of a pre-dimensioned array. The segment of an array can be found as follows:

```
REDIM Array%(0 to 5000)
DestSegment% = VARSEG(Array%(0))
```

The variable **Direction%** decides whether the image will be saved or restored. A value of zero saves the image. Any other value will restore the image.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The memory location must be declared prior to saving the image into the array. To calculate the amount of memory required use the following formula:

```
MemoryNeeded% = ColumnsUsed% * LinesUsed% * 4 + 4
```

Once the amount of memory required has been calculated, you will dimension an integer array with half of the elements contained in MemoryNeeded%. If the value in MemoryNeeded% is greater than 65536, then you must run QB.EXE or QBX.EXE with the /Ah parameter, and compile your programs with this parameter as well. In addition, you will need to create and pass this routine a long integer array where each element will

provide you with 4 bytes of memory space. To make an array covering 128K of memory, dimension it as follows:

```
REDIM LongArray&(0 to 32767)
```

The following example saves and restores a region 10 columns wide by 100 lines high in the upper-left corner of the screen.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

LINE (0, 0) - (79, 99), 1, B
'save the image
MemNeeded% = 10 * 100 * 4 + 4
DIM A%(MemNeeded% \ 2)    'each integer counts for 2 bytes
CALL GMove4VE (1, 0, 10, 100, VARSEG(A%(0)), 0)
WHILE INKEY$ = "": WEND

CLS
'restore the image
CALL GMove4VE (1, 0, 10, 100, VARSEG(A%(0)), -1)
```

By calling VARSEG at the time when GMove4VE is called, you will ensure that if BASIC has moved your array, that you will be passing the proper address of your array to the GMove4VE routine.

The default page for this routine is page 0. Depending upon the value in the parameter Direction%, the default page refers to the destination or the source for the image. The default page can be re-directed using the routine SetGWPages.

A side-effect of this routine is that images saved with the GMove4VE routine can be placed anywhere on the screen using the BASIC PUT statement. However, images captured with the BASIC GET statement cannot be displayed with the GMove4VE routine.

■ **See Also:**

GMove1VE, GMove2VE, GMove3VE, SetGWPages

GScrollVE

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  GScrollVE scrolls a rectangular region on the screen.

- **Syntax:**

```
CALL GScrollVE (BYVAL Col1%, BYVAL Line1%, BYVAL Cols%, _
  BYVAL Lines%, BYVAL ColDelta%, BYVAL LineDelta%)
```

- **Where:**

  Col1% and **Line1%** specify the upper corner of the region to be moved. **Cols%** and **Lines%** specify the size of the region to be moved. These coordinates follow a mixed coordinate window system where column values range from 1 to 80 and line values range from 0 to 479 on a VGA monitor.

  ColDelta% and **LineDelta%** specify the distance the region should be scrolled. A negative number can be used to specify movement in the negative direction for that axis.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

Not only can the window be scrolled left, right, up and down, but diagonal combinations of these are also possible.

The following code fragment scrolls a circle in a diagonal direction on the screen:

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

CIRCLE (120, 120), 40, 15
PAINT (120, 120), 4, 15

FOR T = 1 TO 10
  CALL GScrollVE(1, 0, 22, 169, 1, 8)
NEXT
```

You can use the elements 43 and 44 of the GPDat%() array to check the screens boundaries.

MakeAddressVE%

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

MakeAddressVE% takes an (X, Y) coordinate and returns an EGA or VGA video memory address and a bit pattern. The bit pattern has only one bit set. This bit represents the pixel location within memory.

■ **Syntax:**

```
Address% = MakeAddressVE%(BYVAL XPos%, BYVAL YPos%, BitPattern%)
```

■ **Where:**

XPos% and **YPos%** specify the (X, Y) coordinate.

BitPattern% is a byte value returned from the assembly routine which has one bit set which corresponds to the pixel position in memory.

Comment:

Two parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following example gets the information corresponding to pixel location 100, 200 on the screen.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

Address% = MakeAddressVE% (100, 200)
```

Having a memory location and a bit pattern, the pixel can be set using the DrawByteVE routine. There are more optimal ways to set a pixel, but addressing memory directly is faster than by using X, Y coordinates.

■ **See Also:**

DrawByteVE

MultMonitor%

Assembler
function contained in GW.LIB

- **Purpose:**

  MultMonitor% determines all of the monitors attached to a particular computer. It is possible to have a computer with two or more monitors. This routine will tell you which types of monitors are active.

- **Syntax:**

```
M% = MultMonitor%
```

- **Where:**

  Each bit in the return value represents an adapter and monitor combination which is possible. A value of 0 means no graphics monitor is attached.

BIT	VALUE	MEANING
0	1	Hercules adaptor is attached
1	2	CGA capable adaptor attached
2	4	mono EGA adaptor is attached
3	8	color EGA adaptor is attached
4	16	mono VGA adaptor is attached
5	32	color VGA adaptor is attached
6	64	mono MCGA adaptor is attached
7	128	color MCGA adaptor is attached
8	256	EGA adaptor emulating CGA
9	512	IBM 8514/A adaptor is attached

For example, a system which has both a VGA color monitor and a Hercules monitor connected will return a value of 33 (32 for VGA + 1 for Hercules).

Comments:

To check if a VGA monitor exists, use the following line of code:

```
IF (M% AND 32) <> 0 THEN PRINT "Can use VGA"
```

The file GETVIDEO.BAS tests each bit in the proper order to determine the best monitor available. GETVIDEO.BAS is part of the standard code we suggested in Chapter 1. Also, using the GETVIDEO.BAS and SetVideo combination will help to avoid any misunderstanding of this routine and the values it returns.

PaintBits

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

PaintBits accepts a video memory location, a bit pattern, an old color and a new color. For every occurrence of the old color in that byte of memory, PaintBits replaces it with the new color specified. A "bit pattern" parameter allows you to mask out bits which you don't want re-colored.

■ **Syntax:**

```
CALL PaintBits(BYVAL Col1%, BYVAL Line1%, BYVAL BitPattern%, _
  BYVAL OldColor%, BYVAL NewColor%)
```

■ **Where:**

Col1% and **Line1%** specify the location of a byte in memory. The coordinates follow the mixed coordinate system where column values range from 1 to 80 and line values range from 0 to 479 on a VGA monitor.

BitPattern% is a byte where every bit which is on (a binary value 1) has the ability to be changed.

OldColor% is the color which exists on the screen that we want to change.

NewColor% is the color to replace the former color.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

If you set BitPattern% to 255, this function works exactly like PaintByte which does not accept a bit pattern.

Using this function to change the color back to the original can produce undesirable results. For example, imagine there are three colors in a region: red, white, and blue. If you change the white to a blue, but then want to change it back to white, you will end up changing that which was originally blue to white.

The example below changes every other bit among the first eight horizontal bits on the screen, to blue if and only if they are black.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'    'required for BYVAL's
SCREEN 12                  'sets the monitor in VGA mode

NewColor% = 1              'blue
OldColor% = 0              'black
BitPattern% = 170          'every other bit set
CALL PaintBits (1, 0, BitPattern%, OldColor%, NewColor%)
```

PaintBits' and PaintByte's real power shows up when the region of the screen you want to re-color has colors which exist outside the region, which you do not want to re-color. If it is acceptable for all occurrences of a specified color to be changed, then change its palette instead.

■ **See Also:**

PaintByte

PaintByte

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

PaintByte accepts a byte location on the screen, an old color and a new color. For every occurrence of the old color on the screen, PaintByte replaces it with the new color specified.

■ **Syntax:**

```
CALL PaintByte(BYVAL Col1%, BYVAL Line1%, BYVAL OldColor%, _
  BYVAL NewColor%)
```

■ **Where:**

Col1% and **Line1%** specify the location of a byte in memory. The coordinates follow the mixed coordinate system where column values range from 1 to 80 and line values range from 0 to 479 on a VGA monitor.

OldColor% is the color which exists on the screen that we want to change.

NewColor% is the color to replace the former color.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

Using this function to change the color back to the original can produce undesirable results. For example, imagine there are three colors in a region: red, white, and blue. If you change the white to a blue, but then want to change it back to white, you will end up changing that which was originally blue to white.

The example below changes the first eight horizontal bits on the screen, to red if they are black.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'    'required for BYVAL's
SCREEN 12                  'sets the monitor in VGA mode

NewColor% = 4              'red
OldColor% = 0              'black
CALL PaintBits (1, 0, OldColor%, NewColor%)
```

PaintBits' and PaintByte's real power shows up when the region of the screen you want to re-color has colors which exist outside the region, which you do not want to re-color. If it is acceptable for all occurrences of a specified color to be changed, then change its palette instead.

■ **See Also:**

PaintBits

ScrnDump2

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

ScrnDump2 will print a snapshot of a graphics screen, regardless of the mode, to either an Epson or compatible dot matrix printer, or to a Hewlett Packard LaserJet Series II compatible printer. The images for both types of printers can be printed in either landscape or portrait mode.

■ **Syntax:**

```
CALL ScrnDump2 (DPI$, LptNumber%, Translate%, XMult%, YMult%, _
  LandOrPort%)
```

■ **Where:**

DPI$ indicates the Dots Per Inch resolution when sending to a Hewlett-Packard LaserJet or compatible printer, or is a null string ("") if printing on an Epson FX series or compatible printer. The valid strings for the LaserJet printers are "075", "100", "150" and "300".

LptNumber% is either 1, 2, or 3, to tell ScrnDump2 which parallel printer port to use. If a printer error occurs, ScrnDump2 will return LPTNumber% set to -1.

Translate% is set to any non-zero value to translate all colors on screen to equivalent tile patterns, or it is set to 0 to print all colors as solid black.

XMult% and **YMult%** are aspect multipliers for images which are sent to the LaserJet printers. All aspects for the EPSON style printers are pre-set to give the proper aspect. When printing to an EPSON style printer the values in these variables are left alone by the routine.

LandOrPort% is a true or false variable which determines if the image will be displayed in Landscape or Portrait mode. Any non-zero value will print the image in Landscape mode. A zero value will print in Portrait mode.

Comments:

When printing on a laser printer, ScrnDump2 positions the upper-left corner of the image at the printer's current cursor position.

ScrnDump2 will automatically recognize the current video mode, and determine the number of screen bytes being used and their organization.

An example of printing a VGA high-resolution screen to an Laser Printer at 150 dots per inch in Landscape mode is given below.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

LPTNo% = 1
LandOrPort% = -1
CALL ScrnDump2 ("150", LPTNo%, -1, 1, 1, LandOrPort%)
```

There are many examples of using the ScrnDump2 routine contained in the demonstration program SCRNDUMP.BAS.

There are some restrictions on the valid values for the XMult% and YMult% variables. A buffer inside the ScrnDump2 routine is used to store LaserJet information as it is being flipped into landscape mode. This buffer is only 720 bytes long. Trying to double the size of an image is not always possible. To give a 640 by 200 pixel screen to proper aspect ratio on the HP LaserJet, follow the example below:

```
LptNo% = 1                'Use a variable so we can test it below
CALL ScrnDump2 ("100", LptNo%, -1, 1, 2, -1)
IF LptNo% = -1 THEN PRINT "Error"
```

This will make the image appear like a screen with a 640 by 400 pixel resolution when sent to the printer.

The maximum XMult% and YMult% values are shown in the table below:

SCREEN mode	XMult%	YMult%
1	2	2
2	1	2
3	1	1
7	1	1
8	1	2
9	1	1
10	1	1
11	1	1

SetDestPage

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  SetDestPage sets the destination video page for image moves and for all drawing primitives.

- **Syntax:**

```
CALL SetDestPage (BYVAL DestPage%)
```

- **Where:**

  DestPage% is the destination video page to be set. The first video page is page 0. The EGA has a video page 1, but the VGA does not.

Comments:

The parameter for this routine is passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following code will draw a line on the hidden video page.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in EGA mode

CALL SetDestPage (1)

CALL LineVE (100, 100, 200, 200, 15)
```

The VGA does not have a video page 1, but this routine will mimick the address of an EGA page 1 when used on a VGA.

This routine does not affect any of BASIC's graphics statements.

- **See Also:**

  SetGWPages, SetSourcePage

SetGWPages

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

SetGWPages sets the video pages used by the family of graphics move routines.

■ **Syntax:**

```
CALL SetGWPages (BYVAL SourcePage%, BYVAL DestPage%)
```

■ **Where:**

SourcePage% is the page number for the source of these routines. Valid Page numbers for the EGA are 0 and 1.

DestPage% is the page number for the destination of these routines. Valid Page numbers for the EGA are 0 and 1.

Comments:

Both parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following code fragment makes the Source Page and Destination Page both point to the first video page:

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

CALL SetGWPages (0, 0)
```

This routine does not modify the visual and active pages.

The VGA does not have enough memory to have 2 video pages. Therefore, the only available page is page 0. Using a page value of 1, will not cause an error, but will address memory as if the EGA video mode was active. You can use this knowledge to your advantage with some insight.

It is allowed to make the source and destination pages the same.

The routines SetDestPage and SetSourcePage will allow you to set the video pages individually.

■ **See Also:**

SetDestPage, SetSourcePage

SetLastCoord

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  SetLastCoord sets the coordinate which is the last coordinate drawn to. It does it without modifying any pixel values on screen.

- **Syntax:**

```
CALL SetLastCoord (BYVAL LastX%, BYVAL LastY%)
```

- **Where:**

  LastX% is the X coordinate value to be set.

  LastY% is the Y coordinate value to be set.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The following example sets the coordinate value to (103, 53) and then draw a line 10 pixels long to the right.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

CALL SetLastCoord (103, 53)

CALL LineToStep (9, 0, 15)
```

The benefit that this routine does not affect any of the pixel values on screen, is apparent when the line step routines are used with their logical operators. If the LineStep or LineToStep routines are required this routine will be very useful in implementing the task.

- **See Also:**

  GetLastXCoord%, GetLastYCoord%

SetSourcePage

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  SetSourcePage sets the source video page for graphics moves.

- **Syntax:**

```
CALL SetSourcePage (BYVAL SourcePage%)
```

- **Where:**

  SourcePage% is the source video page to be set. The first video page is page 0. The EGA has a video page 1, but the VGA does not.

- **Comments:**

  The parameter for this routine is passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

  To set the source video page on an EGA screen to page 1, follow the below example.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 9                  'sets the monitor in EGA mode

CALL SetSourcePage(1)
```

  The VGA does not have a video page 1, but this routine will mimick the address of an EGA page 1 when used on a VGA.

  This routine does not affect any of BASIC's graphics statements.

- **See Also:**

  SetDestPage, SetGWPages

SlideDown

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

SlideDown takes a region of the second video page screen and slides it onto the screen at a specified location in a downward direction.

■ **Syntax:**

```
CALL SlideDown(BYVAL ULCol%, BYVAL ULLine%, BYVAL LRCol%, _
  BYVAL LRLine%, BYVAL DestCol%, BYVAL DestLine%)
```

■ **Where:**

ULCol% and **ULLine%** specify the upper-left corner of the source region. **LRCol%** and **LRLine%** specify the lower-right corner of the source region. Together they specify the region to be moved. These coordinates follow the mixed coordinate system where column values range from 1 to 80, and line values range from 0 to 349 on an EGA monitor.

DestCol% and **DestLine%** specify a mixed coordinate value which tells the upper-left corner of the image's final resting place.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

For an example of how to use this routine, see the DEMOFX.BAS example program.

■ **See Also:**

SlideUp, SlideLeft, SlideRight

SlideLeft

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

SlideLeft takes a region of the second video page and slides it onto the screen at a specified location in a horizontal direction towards the left side of the screen.

■ **Syntax:**

```
CALL SlideLeft(BYVAL ULCol%, BYVAL ULLine%, BYVAL LRCol%, _
  BYVAL LRLine%, BYVAL DestCol%, BYVAL DestLine%)
```

■ **Where:**

ULCol% and **ULLine%** specify the upper-left corner of the source region. **LRCol%** and **LRLine%** specify the lower-right corner of the source region. Together they specify the region to be moved. These coordinates follow the mixed coordinate system where column values range from 1 to 80, and line values range from 0 to 349 on an EGA monitor.

DestCol% and **DestLine%** specify a mixed coordinate value which tells the upper-left corner of the image's final resting place.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

For an example of how to use this routine, see the DEMOFX.BAS example program.

■ **See Also:**

SlideDown, SlideUp, SlideRight

SlideRight

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

SlideRight takes a region of the second video page and slides it onto the screen at a specified location in a horizontal direction towards the right side of the screen.

■ **Syntax:**

```
CALL SlideRight(BYVAL ULCol%, BYVAL ULLine%, BYVAL LRCol%, _
  BYVAL LRLine%, BYVAL DestCol%, BYVAL DestLine%)
```

■ **Where:**

ULCol% and **ULLine%** specify the upper-left corner of the source region. **LRCol%** and **LRLine%** specify the lower-right corner of the source region. Together they specify the region to be moved. These coordinates follow the mixed coordinate system where column values range from 1 to 80, and line values range from 0 to 349 on an EGA monitor.

DestCol% and **DestLine%** specify a mixed coordinate value which tells the upper-left corner of the image's final resting place.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

For an example of how to use this routine, see the DEMOFX.BAS example program.

■ **See Also:**

SlideDown, SlideUp, SlideLeft

SlideUp

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

SlideUp takes a region of the second video page and slides it onto the screen at a specified location in an upward direction.

■ **Syntax:**

```
CALL SlideUp (BYVAL ULCol%, BYVAL ULLine%, BYVAL LRCol%, _
  BYVAL LRLine%, BYVAL DestCol%, BYVAL DestLine%)
```

■ **Where:**

ULCol% and **ULLine%** specify the upper-left corner of the source region. **LRCol%** and **LRLine%** specify the lower-right corner of the source region. Together they specify the region to be moved. These coordinates follow the mixed coordinate system where column values range from 1 to 80, and line values range from 0 to 349 on an EGA monitor.

DestCol% and **DestLine%** specify a mixed coordinate value which tells the upper-left corner of the image's final resting place.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

For an example of how to use this routine, see the DEMOFX.BAS example program.

■ **See Also:**

SlideDown, SlideLeft, SlideRight

SplitHorizontal

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

SplitHorizontal takes a region of the second video page and slides it onto the screen at a specified location in a horizontal direction. Every other line of the image will come from the opposite direction. The result is an image which appears to weave itself together.

■ **Syntax:**

```
CALL SplitHorizontal (BYVAL ULCol%, BYVAL ULLine%, BYVAL _
  LRCol%, BYVAL LRLine%, BYVAL DestCol%, BYVAL DestLine%)
```

■ **Where:**

ULCol% and **ULLine%** specify the upper-left corner of the source region. **LRCol%** and **LRLine%** specify the lower-right corner of the source region. Together they specify the region to be moved. These coordinates follow the mixed coordinate system where column values range from 1 to 80, and line values range from 0 to 349 on an EGA monitor.

DestCol% and **DestLine%** specify a mixed coordinate value which tells the upper-left corner of the image's final resting place.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

For an example of how to use this routine, see the DEMOFX.BAS example program.

■ **See Also:**

SlideDown, SlideUp, SlideRight, SlideLeft.

TransferEGA

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

TransferEGA moves any block on any video page to the same location on another video page. This routine is meant for use with EGA adapters.

■ **Syntax:**

```
CALL TransferEGA (BYVAL FromCol%, BYVAL FromLine%, BYVAL _
  Cols%, BYVAL Lines%)
```

■ **Where:**

FromCol% and **FromLine%** specify the upper corner of the region to be moved. **Cols%** and **Lines%** specify the size of the region to be moved. These coordinates follow a mixed coordinate window system where column values range from 1 to 80 and line values range from 0 to 349 on an EGA monitor.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

This routine works like the GMove1VE routine, except that it doesn't have parameters for the destination location on the screen. The destination is assumed to be the same location as on the source page.

The default pages for this routine and others are page 1 for the source information and page 0 for the destination of the image. The source and destination pages can be re-directed using the SetGWPages routine.

The following example copies a full screen image from the second video page to the first video page:

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'   'required for BYVAL's
SCREEN 12                 'sets the monitor in VGA mode

'draw some sample images
LINE (100, 100) - (200, 200), 2, BF
CIRCLE (200, 200), 100, 15
PAINT (200, 200), 2, 15
PCOPY 0, 1                'copy image to second page

CLS
CALL TransferEGA(1, 0, 80, 350)
```

The TransferEGA routine is used by the BASIC DoSegueX routines demonstrated in the example program QSEGUE.BAS.

■ **See Also:**

GMove1VE, GMove2VE, GMove3VE, SetGWPages

Section 4: Mouse Routines

Using A Mouse With A Graphics Mode Application

A Mouse works differently when operating in graphics mode than it does in text mode. The mouse covers, on average, 256 pixels on the screen at any one time. The internal operation of a mouse works as follows:

1) When the mouse cursor is drawn on the screen by the mouse driver, the following occurs:
   a) The square region encompassing where the mouse cursor will be placed is saved by the mouse routines.
   b) A mask is placed over this region which clears (makes black) all pixels that will be beneath the mouse cursor.
   c) The actual mouse cursor is drawn on the screen filling the square region.
2) When the mouse is removed from the screen, the following occurs:
   a) The square region where the mouse is now located will be replaced by the image previously saved.
3) When the mouse is physically moved, the following happens:
   a) The mouse is removed from the screen as described above (under #2).
   b) The mouse is drawn at the new location as described above (under #1).

What concerns programmer is the last of these three mouse functions, especially since the mouse is moved often. The problem occurs if we draw a line (or anything) over the mouse. If the mouse then is moved, the image saved before the line was drawn will be placed on the screen, effectively losing the information we wanted to draw to the screen.

To see the problem, load the demo DEMOMOUS.BAS and move the mouse while it is drawing the boxes. Notice that the mouse disappears and when you move the mouse, a previous color appears in its place. The demo then shows you the proper method for drawing graphics while the

mouse is active. Programming for character mode while the mouse is active is simple; just turn it on and monitor it.

For graphics mode, every, repeat EVERY, time you draw anything, you must temporarily turn off the mouse and turn it on again afterwards. This is done with the HideCursor and ShowCursor routines. These routines communicate with the mouse driver. The mouse driver has a special register which counts the number of times you have called HideCursor. Each time you call ShowCursor the driver decreases this value, and places the mouse cursor on the screen only when this value reaches zero. You will find that it is simpler and more beneficial to call HideCursor as few times as possible, such as at the beginning of a long list of LINE statements. You should be sparing in your use of HideCursor and ShowCursor, because each time you do it, it will take time out of normal screen processing.

For an example of how HideCursor and ShowCursor work, consider the following: You create a routine called DrawPerson which calls a routine called DrawEye. DrawEye can be called directly to make the eye blink, so it has to have a call to HideCursor and ShowCursor inside it. DrawPerson will have already hidden the mouse cursor. This is where the count maintained by the mouse driver is needed. If a total of two calls are made to HideCursor, it will takes two calls to ShowCursor to restore the mouse onto the screen.

ButtonPress

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

ButtonPress will report how many times a specified mouse button was pressed since the last time it was called. It also returns the (X, Y) coordinates where the mouse cursor was located when the button was last pressed.

■ **Syntax:**

```
CALL ButtonPress(Button%, Status%, Count%, X%, Y%)
```

■ **Where:**

Button% is the button of interest, with a 1 indicating button 1, 2 meaning button 2, and 3 for button 3 (if the mouse has a third button).

Status% is the current button status, and has the same meaning as the information returned by the GetCursor mouse routine.

Count% tells how many times the button has been pressed since ButtonPress was last called.

X% and **Y%** hold the mouse cursor position at the time the button was pressed. Use the GetCursor routine to determine the current mouse cursor location.

Comments:

ButtonPress is the only reasonable way to determine when the mouse buttons are active and need attention. Though the GetCursor routine will report the current button status, it would have to be polled repeatedly in a loop.

A good example of implementing ButtonPress can be found in the source code for both PullDownG and VertMenuG.

The comments that accompany the GetCursor routine provide an explanation of interpreting the X and Y values that are returned.

Note that unlike GetCursor, ButtonPress resets the button-press counter to zero each time it is called.

■ **See Also:**

GetCursor

GetCursor

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

GetCursor reports the current location of the mouse cursor and which mouse buttons are currently depressed.

■ **Syntax:**

```
CALL GetCursor(X%, Y%, Status%)
```

■ **Where:**

X% and **Y%** return holding the current mouse cursor coordinates.

Status% is bit-coded to indicate which buttons are currently down.

The button information is represented by bits in the Status% variable, with bit 0 being on to indicate that button 1 is pressed, bit 1 for button 2, and so forth. The various bits may be easily isolated as shown below:

```
CALL GetCursor(X%, Y%, Status%)
IF Status% AND 1 THEN PRINT "Button 1 is pressed"
IF Status% AND 2 THEN PRINT "Button 2 is pressed"
IF Status% AND 4 THEN PRINT "Button 3 is pressed"
```

Comments:

The X% and Y% values returned depend in part on the number of pixels that are available on the screen. This is true even in text mode. As an example, if the mouse cursor is currently at the upper-left corner of the screen, both the X and Y values will be returned as zero. If the cursor moves one character box to the right, the X Value will immediately become eight.

The same thing happens when the cursor is moved downward, in which case the Y value suddenly jumps to eight. Thus, the resolution of a 25-line text screen is considered (to the mouse anyway) as being the same as that of a CGA high-resolution display. The 43 line character mode works the same way but the Y resolution in the 50 line mode jumps in steps of 7.

In graphics modes there are no jumps. If a screen has a resolution of 640 by 350, then the mouse could point to any one of the possible combinations.

Unlike the ButtonPress routine, GetCursor does not reset the count of how many times the buttons have been pressed.

■ **See Also:**

ButtonPress

GrafCursor

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  GrafCursor greatly simplifies defining the shape of the mouse cursor for use in graphics mode.

- **Syntax:**

```
CALL GrafCursor(X%, Y%, Cursor$)
```

- **Where:**

  X% and **Y%** define the cursor "hot spot".

  Cursor$ is either a conventional or fixed length string that contains the new cursor shape. Examples of setting up this string are shown in DEMOMOUS.BAS.

Comments:

The example in the DEMOMOUS.BAS demonstration program shows how GrafCursor can be used with a "pictorial" layout to quickly visualize how the cursor will appear.

The hot spot indicates which pixel the mouse is considered to be on when in graphics mode. Even though the mouse cursor will span several pixels at one time, only one point can be considered to be the actual cursor location. Again, the example in DEMOMOUS.BAS shows this in context.

HideCursor

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  HideCursor turns off the mouse cursor.

- **Syntax:**

```
CALL HideCursor
```

Comments:

Any program that is to be "mouse aware" will need to turn on the mouse cursor before expecting a user to access the mouse. Likewise, it is only common courtesy to turn it off again before returning them to the DOS prompt. Also, for graphics programming, you must turn the mouse off before drawing something on the screen.

One very important point to be aware of regarding the HideCursor routine is how the current on and off status is maintained internally by the mouse driver. Unlike the normal text cursor that is turned on or off with the BASIC LOCATE command, the mouse cursor keeps track of how many times it was turned on or off. Thus, if you call HideCursor twice in a row, you will need to call ShowCursor twice before it will be visible again.

In graphics mode, when you want to draw something at the location of the mouse, it is necessary to turn off the mouse cursor temporarily while you are drawing. In graphics mode, the mouse has a copy of the screen image beneath itself. If you draw over the cursor with the cursor on, when the cursor moves, the mouse driver will re-draw the previous image, without what you drew.

This is why you see the mouse flicker in large graphics applications. These applications turn the mouse off and on many times while drawing to the screen. It is for this reason that the above mentioned characteristic of the HideCursor routine can be useful. If you have multiple routines drawing graphics on the screen, it is necessary that each routine turn the mouse cursor off before drawing and turn it back on before leaving. However, due to the nature of graphics programming, a routine cannot always expect to be called from another routine which has previously turned off the mouse. For example, a routine designed to draw an entire face might call a routine to draw an eye. If the eye routine were to be called separately, it should turn off the mouse cursor itself. If it is called from within another routine which has already turned off the cursor, then it should not turn on the cursor when it is finished. Instead the count maintained by the mouse

driver is merely decremented when the eye routine calls ShowCursor to turn the cursor back on.

■ **See Also:**

ShowCursor.

InitMouse

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

InitMouse is used to determine if a mouse if present in the host PC, and to reset the mouse driver software to its default values.

■ **Syntax:**

```
CALL InitMouse(HaveMouse%)
```

■ **Where:**

HaveMouse% receives -1 if a mouse is present, or 0 if no mouse is installed.

Comments:

Because InitMouse resets the mouse driver values (the mouse cursor color, its travel range and sensitivity, etc.), it would probably be called only once at the start of a program.

Understand that InitMouse doesn't actually detect the physical presence of the mouse hardware. Rather, the mouse driver software must be installed before a mouse will be detected. Newer versions of Microsoft's mouse driver software actually detect if the mouse is physically attached to the machine, and will not load the driver unless the mouse is connected.

Motion

Assembler
subroutine contained in GW.LIB

- **Purpose:**

  Motion allows a program to establish the sensitivity of the mouse cursor motion.

- **Syntax:**

```
CALL Motion(Value%)
```

- **Where:**

  Value% is the desired sensitivity ranging between 1 and 32767, with 1 being the most sensitive.

- **Comments:**

  Even though the mouse driver software allows setting the horizontal and vertical sensitivity separately, Motion uses the same value for both. This seems to be the most logical way to control a mouse, while eliminating yet another passed parameter. If you absolutely must be able to set these values independently, you should use the generic Mouse routine provided with Graphics Workshop like this:

```
CALL Mouse(15, 0, X%, Y%)
```

  Where X% and Y% represent the sensitivity for the X and Y coordinates respectively. 15 represents the mouse service for sensitivity setting.

  The stated upper range for the motion sensitivity is 32767; however, values beyond 100 are hopelessly insensitive.

  You may be interested to know that Microsoft calls the unit of distance for the mouse a "Mickey".

- **See Also:**

  Mouse

Mouse

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

Mouse provides access to all of the mouse services, and is the only way to use those that are not provided in a simplified form with Graphics Workshop.

■ **Syntax:**

```
CALL Mouse(AX%, BX%, CX%, DX%)
```

■ **Where:**

AX% is the number for the mouse service of interest, while **BX%**, **CX%**, and **DX%** assign and return the processor's registers.

Comments:

Mouse provides access to all of the mouse services. Most of the important ones are provided as a simplified call with Graphics Workshop. There may be occasions when you need a mouse capability that we have not included.

For instance, if you want to know how many times the ShowCursor routine needs to be called to make the mouse visible, this information is stored in the mouse's environment. Using mouse service 21, you can find the size of the mouse driver's environment. Then, using mouse service 22, you can retrieve the mouse environment. Since the count used by HideCursor and ShowCursor is not always in the same place you would have to save the mouse environment, make a call to HideCursor (which will change the count) and then retrieve a second copy of the environment. Compare the two environments until you find a byte which is different. During the session, this byte will always be the location of the count and you can retrieve the environment at any time to get the count. This complete example is contained in two routines in Graphics Workshop: GetCount-Location% and GetMouseCount%. An example of using these routines in context is shown in the example program DEMOMOUS.BAS.

■ **See Also:**

GetCountLocation%, GetMouseCount%

SetCursor

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

SetCursor provides a simple way to set a new location for the mouse cursor.

■ **Syntax:**

```
CALL SetCursor(X%, Y%)
```

■ **Where:**

X% and **Y%** represent the new horizontal and vertical positions respectively.

Comments:

The valid X and Y coordinates you specify will depend on the current screen mode. For example, on a CGA graphics screen 1, the acceptable range would be between 0 and 319 for X%, and 0 to 199 for Y%.

ShowCursor

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

ShowCursor turns on the mouse cursor, making it visible. If the cursor is currently visible, ShowCursor does nothing, and leaves the mouse cursor visible.

■ **Syntax:**

```
CALL ShowCursor
```

Comments:

For more information see the comments that accompany the companion routine HideCursor.

■ **See Also:**

HideCursor

Section 5:
Routines from QuickPak Professional and P.D.Q.

We have included several subroutines and functions from QuickPak Professional. These routines are described in this section. The only reason for adding these routines is that they are used by the subroutines VertMenuG and PullDownG to manage information.

AltKey%

Assembler
function contained in GW.LIB

- **Purpose:**

  AltKey% reports if the Alt key is currently depressed.

- **Syntax:**

```
Active = AltKey%
```

- **Where:**

  Active receives -1 if the Alt key is currently down, or 0 if it is not.

Comments:

Because AltKey% has been designed as a function, it must be declared before it can be used. Including the file GWDECL.BAS at the beginning of all your programs will avoid any problems, as it contains a declaration for this function.

AltKey% is designed to return -1 for a true value to also allow the use of the BASIC NOT operator:

```
IF AltKey% THEN . . .
```

or

```
IF NOT AltKey% THEN . . .
```

FindLast%

Assembler
function contained in GW.LIB

- **Purpose:**

  FindLast% scans a conventional (not fixed-length) string array backwards looking for the last non-blank element.

- **Syntax:**

```
NumEls% = UBOUND(Array$)
Last = FindLast%(BYVAL VARPTR(Array$(NumEls%)), NumEls%)
```

- **Where:**

  NumEls% is the number of elements to which Array$() has been dimensioned.

  Last receives the number of the last element that is not empty.

Comments:

Because FindLast% has been designed as a function, it must be declared before it can be used. Including the file GWDECL.BAS at the beginning of all your programs will avoid any problems, as it contains a declaration for this function.

VertMenuG uses this routine to find the actual number of elements in the vertical scrolling menu. The physical array can be larger than the number of elements used. It would be inappropriate for VertMenuG to display blank entries should the array be dimensioned larger than the number of elements is use.

HercThere%

Assembler
function contained in GW.LIB

- **Purpose:**

  HercThere% will report if the QBHERC.COM or MSHERC.COM Hercules graphic support program has been loaded into memory.

- **Syntax:**

```
Loaded% = HercThere%
```

- **Where:**

  Loaded will be set to -1 if QBHERC.COM or MSHERC.COM program is resident, or 0 if it is not.

Comments:

Because HercThere% has been designed as a function, it must be declared before it can be used. Including the file GWDECL.BAS at the beginning of all your programs will avoid any problems, as it contains a declaration for this function.

Note that using this method for doing Hercules graphics is not entirely necessary. The Graphics Workshop routine Screen3 sets the computer in the Hercules high-resolution mode without the need of loading QBHERC.COM or MSHERC.COM. See the routine Screen3 for details.

Even though QuickBASIC supports graphics using a Hercules display adapter, a special TSR (terminate and stay resident) program, MSHERC.COM, must be run first. If this is not done, attempting to use the SCREEN 3 statement to enter graphics mode will cause an "Illegal Function Call" error.

The Graphics Workshop MultMonitor function will tell you if a Hercules display is installed in the host PC, but it does not detect if the necessary support program, MSHERC.COM, has been loaded. This is the purpose of HercThere%.

QuickBASIC 4.0 comes with a program named QBHERC.COM that contains the routines necessary for Hercules graphics. It was renamed to MSHERC.COM when QuickBASIC 4.5 was introduced. HercThere% will detect if either QBHERC.COM or MSHERC.COM is loaded.

- **See Also:**

  Screen3

InStat%

Assembler
function contained in GW.LIB

- **Purpose:**

  InStat% returns the number of characters that are currently pending in the keyboard buffer, without removing them. The PullDownG menu system uses this function to avoid drawing the shadow effect if the user has already pressed an additional keystroke.

- **Syntax:**

```
Count% = InStat%
```

- **Where:**

  Count% receives the number of characters that are currently pending. If any characters are pending, PullDownG will delay drawing the shadow effect.

Comments:

Because InStat% has been designed as a function, it must be declared before it can be used. Including the file GWDECL.BAS at the beginning of all your programs will avoid any problems, as it contains a declaration for this function.

InStat% is very valuable in situations where you need to see if a key is present but do not want to remove it from the keyboard buffer. A good example would be when simulating multi-tasking in a BASIC program.

PDQTimer&

Assembler
function contained in GW.LIB

- **Purpose:**

  PDQTimer& is a better replacement for the BASIC TIMER statement. It works without using floating-point arithmetic, which will save you code space when you compile your programs, since BASIC will not need to load its floating-point math libraries.

- **Syntax:**

```
NumTicks& = PDQTimer&
```

- **Where:**

  PDQTimer& returns a value of the number of clock ticks which occurred in the PC's internal clock since the last reset or overflow.

Comment:

Because PDQTimer& has been designed as a function, it must be declared before it can be used. Including the file GWDECL.BAS at the beginning of all your programs will avoid any problems, as it contains a declaration for this function.

There are 18 ticks per second, so to delay 3 seconds wait 54 ticks. The code fragment below shows how to wait 3 seconds.

```
NumTicks& = PDQTimer& + 54
WHILE NumTicks& > PDQTimer&: WEND
```

Assembly Routines

Chapter 4: BASIC Routines

BASIC Routines

This chapter contains
routines are used el
BASIC routines to a
Many of the routines
the Graphics Manu
routine, can be used

In addition, some o
variables described
Chapter 1. The ben
you are maintaining
not necessary to ma
programming with th
The most common w
routines is Window
coordinates of the u
corner. This works e
statement to draw a D
DisplayBox with di
operator. The XOR
be called a second
on the screen.

```
DEFINT A-Z
DIM GWindow AS W

GWindow.X1% =
GWindow.Y1% = 43
GWindow.X2% = 482
GWindow.Y2% = 30

CALL DisplayBox(GW

WHILE INKEY$ = "": WE

CALL DisplayBox(GW
```

This chapter contains all the routines which are written in BASIC. These routines are used either by the demo programs, or by one of the other BASIC routines to accomplish a task with respect to the graphics screen. Many of the routines provided are here to show how many calls to one of the Graphics Manipulation assembly routines, like the GMove1VE routine, can be used to provide animation.

In addition, some of the BASIC routines take advantage of the TYPE variables described in the "Different Coordinate Systems" section of Chapter 1. The benefit of using these TYPE variables is apparent when you are maintaining information about multiple window regions, and it is not necessary to maintain four sets of variables. A general overview of programming with these window description variables will be given here. The most common variable type used in the Graphics Workshop BASIC routines is Window1. Window1 describes a region by giving the pixel coordinates of the upper-left corner of a region and for the lower-right corner. This works exactly like specifying a region for the BASIC LINE statement to draw a box. For this example we will use the BASIC routine DisplayBox which displays a box on the screen using the XOR logical operator. The XOR operator is used so that the DisplayBox routine can be called a second time to remove the box and restore the original image on the screen.

```
DEFINT A-Z                      'Makes all variables integers

DIM GWWindow as Window1         'defines a local TYPE variable

GWWindow.X1% = 51               'Upper-left X coordinate
GWWindow.Y1% = 43               'Upper-left Y coordinate
GWWindow.X2% = 432              'Lower-right X coordinate
GWWindow.Y2% = 340              'Lower-right Y coordinate

CALL DisplayBox(GWWindow) 'draw the box using XOR

WHILE INKEY$ = "": WEND   'wait for a keystroke

CALL DisplayBox(GWWindow) 'remove the box
```

CircleBAS

BASIC subroutine
contained in CIRCBAS.BAS

■ **Purpose:**

CircleBAS is a BASIC routine which not only explains the algorithm for drawing circles, but also provides a method for drawing circles on a Hercules screen by using the DrawPointH routine to plot pixels. This routine, as shipped, works with the EGA and VGA screen modes. Comments in the routine show how to modify it for Hercules.

■ **Syntax:**

```
CALL CircleBAS (XCenter%, YCenter%, Radius%, CircleColor%, _
  XAspect%, YAspect%)
```

■ **Where:**

XCenter% and **YCenter%** define the center of the circle on the screen.

Radius% is the radius of the circle in pixels.

CircleColor% is the color used to draw the circle.

XAspect% and **YAspect%** are used to draw ellipses.

Comments:

This routine is designed to allow Hercules graphics screens to draw circles using the existing pixel setting routines. An assembly routine, CircleVE, exists and is much faster for drawing circles to EGA and VGA screens. In future versions of Graphics Workshop, there will be assembly routines for drawing circles and lines to the Hercules graphics screen.

■ **See Also:**

DrawPointH, LineBAS

CopyImage

BASIC subroutine
contained in COPYIMAG.BAS

■ **Purpose:**

CopyImage copies a specified block of the graphics screen to another location on the screen. This routine uses the GMove1VE routine to physically move the image.

■ **Syntax:**

```
CALL CopyImage(ULCol, ULLine, LRCol, LRLine, DestCol, DestLine)
```

■ **Where:**

ULCol% and **ULLine%** specify the upper-left corner of the source region. **LRCol%** and **LRLine%** specify the lower-right corner of the source region. Together they specify the region to be moved. These coordinates follow the mixed coordinate system where column values range from 1 to 80 and line values range from 0 to 479 on a VGA monitor.

DestCol% and **DestLine%** specify a mixed coordinate value which tells the upper-left corner of the image's final resting place.

Comments:

The code fragment below demonstrates copying an image 3 columns wide by 20 lines directly to the right of the original image.

```
DEFINT A-Z
'$INCLUDE: 'GWDECL.BAS'
SCREEN 12'sets the monitor in VGA mode

CALL CopyImage (1, 0, 3, 19, 4, 0)
```

For an example of how to use this routine, see the DEMOMOVE.BAS example program.

■ **See Also:**

GMove1VE

Digitize

BASIC subroutine
contained in DIGITIZE.BAS

■ **Purpose:**

Digitize takes a graphics image and builds a lower resolution version of a specified region on the screen. This routine simulates the way features, such as a person's face, are disguised on television.

■ **Syntax:**

```
CALL Digitize(GWWindow AS Window1, Pixels%, Quick%)
```

■ **Where:**

GWWindow is a BASIC TYPE structure. The structure **Window1** is defined in the include file GWDECL.BAS. The variables in this record structure are **X1%**, **Y1%**, **X2%**, and **Y2%**. An example of programming with variables of this kind is shown at the beginning of this chapter.

Pixels% is the size of the boxes created.

Quick% is a boolean variable which, if set to any non-zero value, tells the routine not to do any calculation of the overall color of the region, but to merely use the color which is in the center of the box.

Comments:

A demonstration of this routine can be found in the DEMODIGI.BAS example program.

DisplayBox

BASIC subroutine
contained in XORBOX.BAS

- **Purpose:**

  DisplayBox draws a box outlining the region specified. The box is drawn with exclusive-oring (XOR operator) to retain images already on the screen.

- **Syntax:**

```
CALL DisplayBox(GWWindow AS Window1)
```

- **Where:**

  GWWindow is a BASIC TYPE structure. The structure **Window1** is defined in the include file GWDECL.BAS. The variables in this record structure are **X1%, Y1%, X2%,** and **Y2%.** An example of programming with variables of this kind is shown at the beginning of this chapter.

Comments:

This routine uses LineVE to draw the lines quickly, using the XOR ability of that routine.

A demonstration of this routine can be found in the DEMOBOX.BAS example program.

DisplayBoxFill

BASIC subroutine
contained in XORBOX.BAS

■ **Purpose:**

DisplayBoxFill draws a filled box covering the region specified. The box is drawn with exclusive-oring (XOR operator) to retain images already on the screen.

■ **Syntax:**

```
CALL DisplayBoxFill(GWWindow AS Window1)
```

■ **Where:**

GWWindow is a BASIC TYPE structure. The structure **Window1** is defined in the include file GWDECL.BAS. The variables in this record structure are **X1%**, **Y1%**, **X2%**, and **Y2%**. An example of programming with variables of this kind is shown at the beginning of this chapter.

Comments:

This routine uses LineVE to draw the lines quickly, using the XOR ability of that routine.

A demonstration of this routine can be found in the DEMOBOX.BAS example program.

DisplayPCXFile

BASIC subroutine
contained in DISPLPCX.BAS

■ **Purpose:**

DisplayPCXFile will load the PCX file that is passed to it as a parameter to the specified video page. This routine handles the entire process of loading the image and making the appropriate adjustments to the palette.

■ **Syntax:**

```
CALL DisplayPCXFile(Filename$, VideoPage%)
```

■ **Where:**

Filename$ gives the name of the .PCX graphics file to be loaded. If a .PCX extension is not given it will be appended.

VideoPage% tells it which video page to draw the image on. Unless you have redirected the visual page by using the BASIC SCREEN statement, a value of 0 here will display the image to the currently visible page.

Comments:

The screen mode is not set by this routine if the monitor is presumed to be already in the proper screen mode. It is assumed that when calling this routine, your program will already be using a graphics screen mode. Comments in the code show how this routine can always modify the screen mode.

This routine also calls HandlePCXPalette and WhichPCXScreen to handle interpreting the PCX header information.

■ **See Also:**

HandlePCXPalette, WhichPCXScreen

DisplayPCXFile2

BASIC subroutine
contained in DISPLPC2.BAS

■ **Purpose:**

DisplayPCXFile2 will load the PCX file that is passed to it as a parameter at the specified coordinates on the specified video page. This routine handles the entire process of loading the image and making the appropriate adjustments to the palette.

■ **Syntax:**

```
CALL DisplayPCXFile2 (Filename$, VideoPage%, LineStart%, ColStart%)
```

■ **Where:**

Filename$ gives the name of the .PCX graphics file to be loaded. If a .PCX extension is not given it will be appended.

VideoPage% tells it which video page to draw the image on. Unless you have redirected the visual page by using the BASIC SCREEN statement, a value of 0 here will display the image to the currently visible page.

LineStart% and **ColStart%** are in the Mixed Coordinate System. If both values are 0, no positioning will take place. Also if the image is not an EGA or VGA high-resolution image, no positioning will take place.

Comments:

The screen mode is not set by this routine if the monitor is presumed to be already in the proper screen mode. It is assumed that when calling this routine, your program will already be using a graphics screen mode. Comments in the code show how this routine can always modify the screen mode.

This routine calls upon PositionPCXVE to position the PCX image after the image has been opened with OpenPCXFile%.

This routine also calls HandlePCXPalette and WhichPCXScreen to handle interpreting the PCX header information.

■ **See Also:**

HandlePCXPalette, WhichPCXScreen

DoSegue1

BASIC subroutine
contained in SEGUE1.BAS

- **Purpose:**

  DoSegue1 is used by QuickSegue to transfer the background graphics screen to the visible graphics screen.

- **Syntax:**

```
CALL DoSegue1(SubType%, SegueColor%, Delay%)
```

- **Where:**

  SubType% selects a subtype for the style used in DoSegue1.

SubType%	DESCRIPTION
1	Transfers entire screen instantaneously.
2	Paints entire screen in SegueColor%.

  SegueColor% is the color to place on the screen when the subtype is 2.

  Delay% tells the routine how many milliseconds to delay after each step of the Segue transfer.

- **Comments:**

  This transition cannot be described using a picture. This routine simply makes the entire image appear in one shot.

- **See Also:**

  DoSegue2, DoSegue3, DoSegue4, DoSegue5, DoSegue6.

DoSegue2

BASIC subroutine
contained in SEGUE2.BAS

■ **Purpose:**

DoSegue2 is used by QuickSegue to transfer the background graphics screen to the visible graphics screen.

■ **Syntax:**

```
CALL DoSegue2(SubType%, SegueColor%, Delay%)
```

■ **Where:**

SubType% selects a subtype for the style used in DoSegue2. The larger the value in the variable SubType%, the larger the chuck displayed at each delayed iteration.

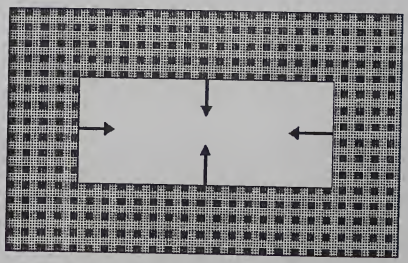
SubType%	DESCRIPTION
1,3,5,7	Paints the screen in the fashion shown below.
2,4,6,8	Brings in the background image in the fashion shown below.

SegueColor% is the color to place on the screen when the subtype is an odd value.

Delay% tells the routine how many milliseconds to delay after each step of the Segue transfer.

Comments:

The picture below shows the direction of the screen coverage for the DoSegue2 routine.



■ **See Also:**

DoSegue1, DoSegue3, DoSegue4, DoSegue5, DoSegue6.

DoSegue3

BASIC subroutine
contained in SEGUE3.BAS

- **Purpose:**

  DoSegue3 is used by QuickSegue to transfer the background graphics screen to the visible graphics screen.

- **Syntax:**

```
CALL DoSegue3(SubType%, SegueColor%, Delay%)
```

- **Where:**

  SubType% selects a subtype for the style used in DoSegue3. The larger values for the variable SubType% select different random patterns for the chunks to fill the screen.

SubType%	DESCRIPTION
1,3,5,7,9,11	Paints the screen in the fashion shown below.
2,4,6,8,10,12	Brings in the background image in the fashion shown below.

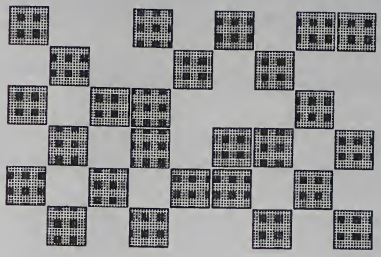
  Each of the higher values produces a different random display.

  SegueColor% is the color to place on the screen when the subtype is an odd value.

  Delay% tells the routine how many milliseconds to delay after each step of the Segue transfer.

Comments:

The picture below shows how the screen might look part way through the execution of this routine. Some of the blocks in the image have appeared and others are still missing.



■ **See Also:**

DoSegue1, DoSegue2, DoSegue4, DoSegue5, DoSegue6.

DoSegue4

BASIC subroutine
contained in SEGUE4.BAS

■ **Purpose:**

DoSegue4 is used by QuickSegue to transfer the background graphics screen to the visible graphics screen.

■ **Syntax:**

```
CALL DoSegue4(SubType%, SegueColor%, Delay%)
```

■ **Where:**

SubType% selects a subtype for the style used in DoSegue4. The larger the value in the variable SubType%, the larger the chuck displayed at each delayed iteration.

SubType%	DESCRIPTION
1,3,5,7,...	Paints the screen in the fashion shown below.
2,4,6,8,...	Brings in the background image in the fashion shown below.

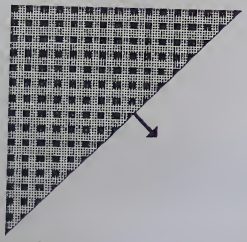
The dots mean that this routine can use higher values if you so choose. For higher values, the blocks will become very large.

SegueColor% is the color to place on the screen when the subtype is an odd value.

Delay% tells the routine how many milliseconds to delay after each step of the Segue transfer.

Comments:

The picture below shows the direction of the screen coverage for the DoSegue4 routine.



■ **See Also:**

DoSegue1, DoSegue2, DoSegue3, DoSegue5, DoSegue6.

DoSegue5

BASIC subroutine
contained in SEGUE5.BAS

■ **Purpose:**

DoSegue5 is used by QuickSegue to transfer the background graphics screen to the visible graphics screen.

■ **Syntax:**

```
CALL DoSegue5(SubType%, SegueColor%, Delay%)
```

■ **Where:**

SubType% selects a subtype for the style used in DoSegue5. The larger the value in the variable SubType%, the larger the number of lines growing for each iteration.

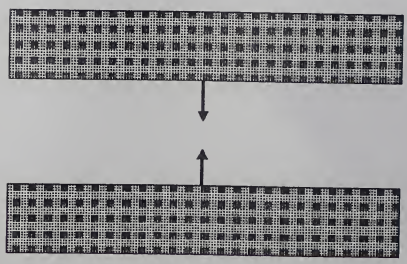
SubType%	DESCRIPTION
1,3,5,7	Paints the screen in the fashion shown below.
2,4,6,8	Brings in the background image in the fashion shown below.3

SegueColor% is the color to place on the screen when the subtype is an odd value.

Delay% tells the routine how many milliseconds to delay after each step of the Segue transfer.

Comments:

The picture below shows the direction of the screen coverage for the DoSegue5 routine.



■ **See Also:**

DoSegue1, DoSegue2, DoSegue3, DoSegue4, DoSegue6.

DoSegue6

BASIC subroutine
contained in SEGUE6.BAS

■ **Purpose:**

DoSegue6 is used by QuickSegue to transfer the background graphics screen to the visible graphics screen.

■ **Syntax:**

```
CALL DoSegue6(SubType%, SegueColor%, Delay%)
```

■ **Where:**

SubType% selects a subtype for the style used in DoSegue6. The larger the value in the variable SubType%, the larger the chuck displayed at each delayed iteration.

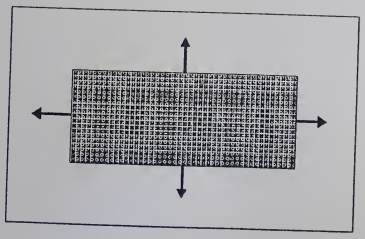
SubType%	DESCRIPTION
1,3,5,7	Paints the screen in the fashion shown below.
2,4,6,8	Brings in the background image in the fashion shown below.

SegueColor% is the color to place on the screen when the subtype is an odd value.

Delay% tells the routine how many milliseconds to delay after each step of the Segue transfer.

Comments:

The picture below shows the direction of the screen coverage for the DoSegue6 routine.



■ **See Also:**

DoSegue1, DoSegue2, DoSegue3, DoSegue4, DoSegue5.

Draw3DButton

BASIC subroutine
contained in BUTTON.BAS

■ **Purpose:**

Draw3DButton draws a three-dimensional button on screen at the specified location.

■ **Syntax:**

```
CALL Draw3DButton(XPos%, YPos%, Wdth%, Height%, ThirdDimension%, Colr%)
```

■ **Where:**

XPos% and **YPos%** specify the upper-left corner of the three-dimensional buttons.

Wdth% and **Height%** specify the size of the button in pixels.

ThirdDimension% specifies the number of pixels involved in giving the button its three-dimensional look.

Colr% is the color for the face of the button. The top of the button uses a bright white highlight, and the bottom of the button uses a black shadow color.

Comments:

The routine could be modified to give different highlight and shadow colors.

DrawCursor

BASIC subroutine
contained in CURSOR.BAS

■ **Purpose:**

DrawCursor draws a graphics cursor on the screen using the XOR logical operation at the specified location.

■ **Syntax:**

```
CALL DrawCursor(XPos%, YPos%, Wdth%, Height%, OnOrOff%, Timing%)
```

■ **Where:**

XPos% and **YPos%** specify the upper-left coordinate to start drawing the cursor. These positions are absolute pixel locations on the screen.

Wdth% specifies the horizontal width of the cursor in pixels.

Height% specifies the height of the cursor.

OnOrOff% specifies whether the cursor is currently visible on the screen. A value other than zero means that the cursor is visible.

Timing% holds the time (in 1/18ths of a second) between blinking the cursor on and blinking the cursor off. A value of 9 would produce a complete cycle once every second.

Comments:

DrawCursor uses the LineVE routine from Graphics Workshop to draw the Exclusive-ORed cursor on the screen.

A demonstration of DrawCursor can be found in the example program DEMOCURS.BAS. The GEditor routine also uses DrawCursor, and is demonstrated in the example program DEMOEDIT.BAS.

■ **See Also:**

GEditor

DrawText

BASIC subroutine
contained in DRAWTEXT.BAS

■ **Purpose:**

DrawText is used to draw a text string using the fonts available with GraphPak Professional.

■ **Syntax:**

```
CALL DrawText(X%, Y%, Text$, Angle%, Colr%, SizeMultiplier#)
```

■ **Where:**

The variables **X%** and **Y%** are pixel positions of the upper-left corner of the first character to be drawn.

Text$ is a string of the phrase to be drawn.

Angle% is the angle specified in degrees at which the text is to be drawn. An angle of 0 draws text straight across the screen.

Colr% is the color of the text. Both DrawText and StepText phrases can have shadows drawn underneath them. Adding 128 to the color will activate the shadow effect.

SizeMultiplier# is the size of the font. For example, 1# = same size as the original definition of the font, 2# = twice as big, .75 = 3/4 the size of the font.

Comments:

The color for the shadow effect mentioned above can be changed by modifying the variable GPDat%(14). See Appendix C for information about the GPDat%() array.

The GraphPak fonts are used throughout the demonstration programs, and particularly in the example program DEMOFONT.BAS.

■ **See Also:**

GetWidth%, StepText

FullZoom

BASIC subroutine
contained in ZOOM.BAS

■ **Purpose:**

FullZoom zooms in on a portion of the screen. The algorithm takes the window size and calculates the proportions necessary to zoom the image to the full scale.

■ **Syntax:**

```
CALL FullZoom(GWWindow AS Window1)
```

■ **Where:**

GWWindow is a BASIC TYPE structure. The structure **Window1** is defined in the include file GWDECL.BAS. The variables in this record structure are **X1%, Y1%, X2%,** and **Y2%.** An example of programming with variables of this kind is shown at the beginning of this chapter.

Comments:

The FullZoom routine is demonstrated in the example program DEMOZOOM.BAS.

GEditor

BASIC subroutine
contained in GEDITOR.BAS

■ **Purpose:**

GEditor is a graphics mode text input routine that also allows editing an existing string. GEditor is designed for use with the VGA and EGA high-resolution screen modes.

■ **Syntax:**

```
CALL GEditor(Edit$, LeftCol%, KeyCode%, TxtPos%)
```

■ **Where:**

Edit$ is the string being entered or edited, which must be pre-assigned to the correct maximum length (see below).

LeftCol% is the column number used to place the cursor when editing the string. LeftCol% is maintained in this fashion to take advantage of the re-entrant abilities of the GEditor routine.

KeyCode% indicates how editing was terminated. It returns the ASCII code of the last character typed, or it returns the negative value of an extended character if an extended character was typed.

TxtPos% is the current position of the cursor within the edited string.

Comments:

If you want to restrict the length of the input to 16 characters, then you would specify an Edit$ with 16 spaces as in the example below:

```
Edit$ = SPACE$(16)
CALL GEditor(Edit$, LeftCol%, KeyCode%, TxtPos%)
```

If Edit$ already contains information, you would pad the string with blanks as in the example below:

```
Edit$ = Edit$ + SPACE$(16 - LEN(Edit$))
```

A complete demonstration of the GEditor routine is given in the example program DEMOEDIT.BAS.

The GEditor routine uses the DrawCursor routine to give the user a cursor while in graphics mode.

■ **See Also:**

DrawCursor

GetCountLocation%

BASIC function
contained in MOUSECNT.BAS

■ **Purpose:**

GetCountLocation% obtains the location within the mouse driver's environment of where the count of how many times the mouse cursor has been turned off is located. This can be useful in telling you whether or not the mouse cursor is currently visible, or telling you how many times a call to ShowCursor is required to make the cursor visible once again.

■ **Syntax:**

```
CountPosition% = GetCountLocation%
```

Comments:

Because GetCountLocation% has been designed as a function, it must be declared before it can be used. Including the file GWDECL.BAS at the beginning of all your programs will avoid any problems, as it contains a declaration for this function.

This function tells only the location of the variable in the mouse environment which will give you this information. A call to the function GetMouseCount% is required to actually determine whether or not the mouse cursor is visible.

We suggest that you use this routine immediately after setting your graphics screen mode, as the mouse environment does change after changing screen modes. We also suggest you execute this function just once, as it does take time and valuable data space to perform this operation.

A complete demonstration of this routine can be found in the example program DEMOMOUS.BAS.

■ **See Also:**

GetMouseCount%

GetMouseCount%

BASIC function
contained in MOUSECNT.BAS

Purpose:

GetMouseCount% tells you how many times the HideCursor routine has been called. If the mouse is currently visible then this routine will return a zero for the number of times the HideCursor routine has been called.

Syntax:

```
MouseCount% = GetMouseCount%(CountPosition%)
```

Where:

CountPosition% is the position of the count variable within the mouse driver's environment. To get the CountPosition%, a call to the function GetCountLocation% is made as shown in the example below:

```
CountPosition% = GetCountLocation%
```

Comments:

Because GetMouseCount% has been designed as a function, it must be declared before it can be used. Including the file GWDECL.BAS at the beginning of all your programs will avoid any problems, as it contains a declaration for this function.

To set up the mouse driver and check to see if the mouse cursor is active, execute the following code:

```
DEFINT A-Z                'Good practice for all programs
'$INCLUDE: 'GWDECL.BAS'
SCREEN 9 'optional, as GetMouseCount% works
                          'in all screen modes

CALL InitMouse(There%)    'Initialize the mouse
IF NOT There% THEN PRINT "No Mouse Driver": END

CountPosition% = GetCountLocation%
IF GetMouseCount%(CountPostion%) = 0 THEN
  PRINT "Mouse visible"
ELSE
  PRINT "Mouse not visible"
ENDIF
```

A complete demonstration of the routine can be found in the example program DEMOMOUS.BAS

See Also:

GetCountLocation%

GetOutlineWidth%

BASIC function
contained in OUTLTEXT.BAS

- **Purpose:**

  GetOutlineWidth% determines the width in pixels of a text string using the vector font widths of the currently loaded vector font.

- **Syntax:**

```
Width% = GetOutlineWidth%(Text$)
```

- **Where:**

  Text$ is a string which holds the phrase to be drawn.

Comments:

Because GetOutlineWidth% has been designed as a function, it must be declared before it can be used. Including the file GWDECL.BAS at the beginning of all your programs will avoid any problems, as it contains a declaration for this function.

The width is for a font size of 1. If you will be drawing the text at any other size, simply multiply that size by the width of the text string to obtain the actual number of pixels that will be used.

Centering a Graphics Workshop font can be done by using the screen width variable GPDat%(43) and the value returned by this routine.

```
Start% = (GPDat%(43) - GetOutlineWidth%(Text$)) \ 2
CALL OutLineText(Start%, Y%, Text$, Angle%, Colr%, Mult%, Divid%)
```

This takes half of the screen size as being the center and subtracts half of the total width of the string in pixels. This will give you the X position to start drawing the string, in order to have it appear centered.

- **See Also:**

  OutlineText

GetWidth%

BASIC function
contained in DRAWTEXT.BAS

■ **Purpose:**

GetWidth determines the width in pixels of a text string using the font widths of the currently loaded GraphPak font.

■ **Syntax:**

```
Width% = GetWidth%(Text$)
```

■ **Where:**

Text$ is a string which holds the phrase to be drawn.

Comments:

Because GetWidth% has been designed as a function, it must be declared before it can be used. Including the file GWDECL.BAS at the beginning of all your programs will avoid any problems, as it contains a declaration for this function.

The width is for a font size of 1. It must be multiplied by the font size that is going to be used in order to obtain the width in pixels of the string you will actually be placing on the screen.

Centering a GraphPak font can be done by using the screen width variable GPDat%(43) and the value returned by this routine.

```
Start% = (GPDat%(43) - GetWidth%(Text$)) \ 2
CALL DrawText(Start%, Y%, Text$, Angle%, Colr%, TextSize#)
```

This takes half of the screen size as being the center, and subtracts half of the total width of the string in pixels. This will give you the X position to start drawing the string, in order to have it appear centered.

■ **See Also:**

DrawText, StepText

GPaintBox

BASIC subroutine
contained in GPAINTBX.BAS

- **Purpose:**

  GPaintBox changes all occurrences of one color to another color within a rectangular region of the screen.

- **Syntax:**

```
CALL GPaintBox(ULCol, ULLine, LRCol, LRLine, OldColor, NewColor)
```

- **Where:**

  ULCol% and **ULLine%** specify the upper-left corner of the region. **LRCol%** and **LRLine%** specify the lower-right corner of the region. Together they specify the region to be re-colored. These coordinates follow the mixed coordinate system where column values range from 1 to 80 and line values range from 0 to 479 on a VGA monitor.

  OldColor% is the color you want to have changed. If OldColor is set to -1, the value in GPDat%(57) is used.

  NewColor% is the new color to replace the former color.

Comment:

GPDat%() element 57 holds the background color for the screen. This is a value you must set to specify the overall background color of the screen. If you specify a -1 for OldColor% and GPDat%(57) is zero, GPaintBox will use the color in the upper-left most corner of the region.

GPaintBox is used by the menu routines PullDownG, VertMenuG, and MsgBoxG to display a shadow effect. The shadow effect produced by this routine modifies only the background color for the screen.

- **See Also:**

  PullDownG, VertMenuG, MsgBoxG

HandlePCXPalette

BASIC subroutine
contained in PCXHEADR.BAS

■ **Purpose:**

HandlePCXPalette takes the palette information from the PCX image and sets the screens palette.

■ **Syntax:**

```
CALL HandlePCXPalette(Array$, WhichScreen%)
```

■ **Where:**

Array$ contains all the header information for the PCX file. A description of the PCX header information is contained in Appendix A.

WhichScreen% tells the routine the intended screen mode for the PCX image. HandlePCXPalette uses this to determine how to communicate the palette information to the various hardware configurations.

Comments:

To get a PCX header into Array$, the routine OpenPCXFile% is used. An example of using the routine HandlePCXPalette can be found in the example program VIEWPCX.BAS, or in the routine DisplayPCXFile.

This routine handles the different idiosyncracies of the PCX palette. Some PCX files do not have palette information in them, and therefore use the standard palette.

For most cases, all of the palette information is for the EGA and VGA modes. The EGA has a maximum palette of 16 colors out of 64 choices. The VGA has a maximum palette of 256 colors out of approximately 256,000. Note the VGA palette numbers are not necessarily sequential. See Appendix B for more on palettes.

In the case of a VGA image being displayed on an EGA display, this routine attempts to map the VGA's large palette into the EGA's limited palette.

■ **See Also:**

OpenPCXFile%

Interlude1

BASIC subroutine
contained in INTER1.BAS

- **Purpose:**

  Interlude1 places a Movie Director's Clicker on the screen with a message of your choice. This routine can be used by the QuickSegue program, or it can be added to any program to display a title.

- **Syntax:**

```
CALL Interlude1(Text$)
```

- **Where:**

  Text$ is a string which can have 1 or more phrases separated by underscore characters. Underscore characters tell the routine to start a new line.

- **Comments:**

  The font used by this routine can be specified using the SetGWFont routine before calling this routine.

  The fonts will size themselves automatically to fit inside the Clicker. If you are using many lines of text and this routine makes a line with too large a font for your tastes, add an equal amount of spaces to each side of the text for that line. This will force this routine into believing it needs to fit more text on the line and therefore create a smaller font. Try something like below:

```
Text$ = "Graphics_          Made          _Easy"
```

  Because Graphics Workshop uses a proportional font system, spaces do not take up much in the size of a font so you may want to add many of them.

- **See Also:**

  SetGWFont, Interlude2.

Interlude2

BASIC subroutine
contained in INTER2.BAS

■ **Purpose:**

Interlude2 moves rectangular boxes across the screen in random directions. The boxes move beneath all items on the screen which are marked as being in the foreground. This routine adds motion to an otherwise static display.

■ **Syntax:**

```
CALL Interlude2(TextColors%(), MaxBoxes%, Seconds%)
```

■ **Where:**

TextColors%() is an array of colors which have been used as foreground colors. Any items using these colors will be considered in the foreground of the screen. You use this to identify which objects are part of your picture. All other colors (0 - 15) will be valid for use by the routine in creating its rectangular boxes.

MaxBoxes% is the maximum number of moving boxes on screen at any one time. Use this to control the speed of the moving boxes. If there are too many boxes (i.e. 20) they will move slowly due to the overhead of moving so many boxes.

Seconds% is the number of seconds for the routine to run. This will be dependent upon the speed of any boxes left on screen at the end of **Seconds%** seconds.

Comments:

The routine may run for more seconds than were requested, as it waits until all boxes have exited the screen before exiting. The average box could take as long as five seconds to cross the entire screen, so as many as five seconds might be added to the time this routine executes.

The box sizes are random and range from 20 pixels to 40 pixels wide, and 80 pixels to 200 pixels in length. In addition some boxes move faster than others because they skip a line as they move across the screen.

■ **See Also:**

Interlude1.

LineBAS

BASIC subroutine
contained in LINEBAS.BAS

■ **Purpose:**

LineBAS demonstrates the algorithm used for drawing lines, and gives a template for creating a line routine which will work with the Hercules screen mode by using DrawPointH to plot pixels. The routine, as shipped, uses the EGA and VGA screen modes, but has comments on how to change it to use the Hercules graphics screen.

■ **Syntax:**

```
CALL LineBAS (x1%, y1%, x2%, y2%, LineColor%)
```

■ **Where:**

Coordinate pairs (**x1%, y1%**) and (**x2%, y2%**) are within the range of the screen.

The **LineColor%** is the color of the line.

Comments:

This routine is designed only to allow Hercules graphics screens to draw lines using the existing pixel setting routines. An assembly routine, LineVE, exists and is much faster for drawing circles to EGA and VGA screens. In future versions of Graphics Workshop, there will be assembly routines for drawing lines and circles to the Hercules graphics screen.

■ **See Also:**

CircleBAS, DrawPointH

LoadFont

BASIC subroutine
contained in DRAWTEXT.BAS

■ **Purpose:**

LoadFont loads the specified GraphPak font file into a Font$() array for later use by the text drawing routines, DrawText and StepText. Multiple font definitions can be loaded at any point in time.

■ **Syntax:**

```
CALL SetGPFont(1)
CALL LoadFont(FontFile$)
```

■ **Where:**

FontFile$ is a filename for one of the GraphPak style fonts. This string may include a path if desired.

Comments:

The file extension ".GFN" is assumed and automatically added to each filename passed to the routine. A standard font file is HELV12.GFN. A list of other font files is located at the end of this chapter.

The call to SetGPFont tells the LoadFont routine which number to identify with the font we are loading. All loaded fonts remain in memory. The font selected by SetGPFont will be the font used by the next calls to DrawText or StepText.

■ **See Also:**

SetGPFont, DrawText, StepText

LoadOutlineFont

BASIC subroutine
contained in OUTLTEXT.BAS

■ **Purpose:**

LoadOutlineFont loads the specified Graphics Workshop Vector font file specified into a OutlineFont$() array for later use by the text drawing routines, OutlineText. Multiple vector font definitions can be loaded at any point in time.

■ **Syntax:**

```
CALL SetGWFont(1)
CALL LoadOutlineFont(FontFile$)
```

■ **Where**

FontFile$ is a filename for one of the Graphics Workshop Vector fonts. This string may include a path if desired.

Comment:

The file extension of ".QFN" is automatically assumed and added to each filename passed to it. A standard outline font is HELV.QFN. A list of all available font files is located at the end of this chapter.

The call to SetGWFont tells the LoadOutlineFont routine which number to identify with the Vector font we are loading. All loaded fonts remain in memory, and can be selected with the SetGWFont routine at any time.

■ **See Also:**

SetGWFont, OutlineText

LtsMenuG

BASIC subroutine
contained in LTSMENU.BAS

Purpose:

The LtsMenuG routine provides a single-line Lotus 1-2-3 "look alike" menu system. A list of choices is displayed horizontally on a single line. A choice can be made either by using the arrow keys and pressing Enter, or by pressing a key that corresponds to the first letter of a choice.

Syntax:

```
CALL LtsMenuG (Item$(), Choice%)
```

Where

Item$() is a conventional string array containing a list of the menu items. The maximum length for any menu item is 78 characters.

Choice% returns the selection made by the user. If the user presses Esc, then Choice% will return a 0.

Comment:

As shipped, LtsMenuG waits until the Enter key is pressed before returning to the calling program. However, comments in the source code show how to have it return as soon as the first letter of a choice has been pressed. If you make this modification, though, be aware that each choice must begin with a unique first letter.

LtsMenuG recognizes the Home and End keys and places the cursor on the first and last menu item respectively.

The example program DEMOLTS.BAS shows this routine in use.

See Also:

Lts2MenuG

Lts2MenuG

BASIC subroutine
contained in LTSMENU.BAS

■ **Purpose:**

The Lts2MenuG routine provides a double-line Lotus 1-2-3 "look alike" menu system. A list of choices is displayed horizontally on a single line. The second line is used to provide an explanation of the currently selection menu item. A choice can be made either by using the arrow keys and pressing Enter, or by pressing a key that corresponds to the first letter of a choice.

■ **Syntax:**

```
CALL Lts2MenuG (Item$(), Prompt$(), Choice%)
```

■ **Where**

Item$() is a conventional string array containing a list of the menu items.

Prompt$() is a parallel string array containing the help or explanation of each menu item relating to the items in the Item$() array.

Choice% returns the selection made by the user. If the user presses Esc, then Choice% will return a 0.

Comment:

As shipped, Lts2MenuG waits until the Enter key is pressed before returning to the calling program. However, comments in the source code show how to have it return as soon as the first letter of a choice has been pressed. If you make this modification, though, be aware that each choice must begin with a unique first letter.

Lts2MenuG recognizes the Home and End keys and places the cursor on the first and last menu item respectively.

The example program DEMOLTS.BAS shows this routine in use.

■ **See Also:**

LtsMenuG

MsgBoxG

BASIC subroutine
contained in MSGBOX.BAS

■ **Purpose:**

MsgBoxG provides a quick and attractive way to display a message, with word wrap automatically centered on the screen. The underlying screen is always saved and it may be restored again later. The message box uses a shadow effect to simulate depth.

■ **Syntax:**

```
CALL MsgBoxG(Message$, Wide%)
```

■ **Where:**

**Message$** is a single continuous string to be displayed. If Message$ is null, the most recently displayed message is cleared, and the underlying screen restored.

Wide% is the desired width of the text (up to 74).

Comments:

The top line of the MsgBoxG is placed at the current cursor line, so LOCATE should be used to set that before you call MsgBoxG.

When MsgBoxG is called, the first thing it does is check the length of the message string. If it is not null, it first saves the underlying screen and then displays the message. To clear the message and restore the original screen, simply call MsgBoxG again with a null string.

Be sure that you don't call MsgBoxG with a null string, unless it has already been called at least once before. Also be aware that the message should always be cleared before a new one is displayed. Otherwise, there will be no way to restore the original screen.

The width is limited to 74 because MsgBoxG draws a border around the text, and adds an extra blank space to make the text easier to read. Two additional columns are needed to accommodate the shadow.

All of the colors for this routine are defined in the GPDat%() array. Their values are all initialized in the file GETVIDEO.BAS. The benefit of isolating the color definitions to a single file is that you may customize them to your own preferences, and they will then be reflected in all of the programs that use GETVIDEO.BAS.

MsgBoxG is shown in context in the DEMOMENU.BAS example program.

NightFall

BASIC subroutine
contained in FADE.BAS

■ **Purpose:**

NightFall simulates a gradual nightfall, by bringing random portions of the second video page to the current active video page. Images fade in from the top of the screen towards the bottom. This routine uses the FadeEGA routine to accomplish the fading effect.

■ **Syntax:**

```
CALL NightFall
```

Comments:

It is advisable to use this routine and this effect when the overall background of the two screen images is going to be similar. Any time when there is a large portion of the screen which will not be changed by this routine, the effect is much more pleasing to the eye and doesn't appear rigid. See the demo program DEMOFADE.BAS for a visual description of this routines' functioning.

OutlineText

BASIC subroutine
contained in OUTLTEXT.BAS

- **Purpose:**

  OutlineText draws a string using the Graphics Workshop Vector fonts.

- **Syntax:**

```
CALL OutlineText(X%, Y%, Text$, Angle%, Colr%, Mult%, Divide%)
```

- **Where:**

  The **X%** and **Y%** values represent the pixel coordinates for the upper-left corner of the first character to be drawn.

  Text$ is the string of the phrase to be drawn.

  Angle% is the angle in degrees at which the text will be drawn. An Angle% of 0 will draw text in the normal text direction.

  Colr% is the color 0-15 for the text to drawn.

  Mult% is an integer multiplier for the size of the text.

  Divide% is an integer divider for the size of the text.

Comments:

To use the Mult% and Divide% parameters to obtain a font size of 3/4 of the original font definition, set Mult% to 3 and Divide% to 4. This allows the routine to use integer arithmetic. See Chapter 6 for more information on using the Mult% and Divide% variables to obtain any size font.

A companion function, GetOutlineWidth%, can be used to determine the length in pixels of the Text$ before it is drawn to the screen. This can be of great use in the positioning of a text string.

- **See Also:**

  GetOutlineWidth%

PCXCAP

BASIC program
contained in PCXCAP.BAS

■ **Purpose:**

PCXCAP is TSR utility for capturing PCX images from just about any graphics mode program. To use PCXCAP, run PCXCAP.EXE from the command line, and the at a time when you are on a graphics screen, that you want to capture, press Alt-S and then type in an 8 letter filename.

■ **Syntax:**

```
PCXCAP
```

■ **Where:**

This is what you would type at the command line to start the utility. Type it a second time at the command line to remove it from memory.

Comments:

Although complete source code is provided for your amusement, it requires that you have another of our products, P.D.Q to recompile it.

PCXCAP calls upon the SavePCX??? routines to actually save the PCX image from the screen. Only in the EGA and VGA modes can you save only a portion of the image. In this case PCXCAP calls upon SavePCXRegionVE to save only a portion of the screen.

PositionBox

BASIC subroutine
contained in XORBOX.BAS

■ **Purpose:**

PositionBox is a complete routine for placing a box selector on screen and accepting user input to move and change the size of the box. The routine looks for cursor keys and selects a rectangular region on the screen using the XOR ability of the LineVE routine.

■ **Syntax:**

```
CALL PositionBox(GWWindow AS Window1, Style%, EscPressed%)
IF ESCPressed%  0 THEN PRINT "Esc was pressed"
```

■ **Where:**

GWWindow is a BASIC TYPE structure. The structure **Window1** is defined in the include file GWDECL.BAS. The variables in this record structure are **X1%, Y1%, X2%,** and **Y2%.** An example of programming with variables of this kind is shown at the beginning of this chapter.

Style% is a boolean variable. If it is 0, the outline of a box will be used; otherwise a solid region is used to display the area.

ESCPressed% is used to determine whether or not the Esc key was pressed.

Comments:

The keys accepted by this routine are any of the **Cursor** keys, **Enter**, **SpaceBar**, and **Esc**. The routine remains active until the **Esc** or **Enter** key is pressed.

The **Cursor** keys move the active corner.

The **SpaceBar** toggles which corner is active, either the upper-left or lower-right corner. The routine starts with the upper-left corner.

The **Enter** key accepts the box position as it is on the screen. The new values can be found in the GWWindow array.

The **Esc** key stops use of the routine. A value of -1 is returned in EscPressed% if the Esc key was pressed.

PullDownG

BASIC subroutine
contained in PULLDOWN.BAS

■ **Purpose:**

PullDownG is a complete graphics mode multiple-menu subprogram with many important capabilities including full support for a mouse. Besides being able to display more than one list of choices, it also always saves the underlying screen and accommodates a separating divider between related groups of items.

Furthermore, selected menu items may be allowed or disallowed at will. Finally, PullDownG may be operated in a unique multi-tasking mode, whereby it is polled periodically to see if a choice has been selected.

■ **Syntax:**

```
CALL PullDownG(Choice$(), Status%(), Menu%, Choice%, Ky$, Action%)
```

■ **Where:**

Choice$() is a two-dimensional array containing the list of choices for each menu. If any element contains a hyphen only ("-"), it will be replaced by a separating line and will not be selectable by the user.

Status%() is a parallel, two-dimensional array that indicates which choices are active. Choices can be deactivated by assigning a non-zero value to the element that corresponds to a given item in the Choice$() array.

Menu% indicates which menu was active when a choice was selected, and may also be pre-loaded to force a given menu to be displayed initially.

Choice% indicates which choice was selected, and may also be pre-loaded to force a given choice to be highlighted initially.

Ky$ holds the last key that was pressed by the user. This is used to determine if the user pressed <Esc> to exit the menu system.

Action% tells PullDownG how it is being used. The possible values for the Action% parameter are discussed in the section entitled "Multi-Tasking Menus" in Chapter 1.

Comments:

All of the colors for this routine are defined in the GPDat%() array. Their values are all initialized in the file GETVIDEO.BAS. The benefit of isolating the color definitions to a single file is that you may customize

them to your own preferences, and they will then be reflected in all of the programs that use GETVIDEO.BAS.

PullDownG is explained in depth in the section entitled "Multi-Tasking Menus" in Chapter 1, and two complete demonstrations are also provided. DEMOPULL.BAS shows the minimum setup required for calling PullDownG, and DEMOMENU.BAS illustrates some of its more advanced uses.

■ **See Also:**

PullDnMSG

PullDnMSG

BASIC subroutine
contained in PULLDNMS.BAS

■ **Purpose:**

PullDnMSG is a complete graphics mode multiple-menu subprogram with many important capabilities including full support for a mouse. Besides being able to display more than one list of choices, it also always saves the underlying screen and accommodates a separating divider between related groups of items. The PullDnMSG routine mimics as closely as possible the functionality of the Microsoft Windows menuing system. Each of the menu items has a hotkey assigned to it. The hotkey is then underlined by the PullDnMSG routine when shown on screen.

Furthermore, selected menu items may be allowed or disallowed at will. Finally, PullDnMSG may be operated in a unique multi-tasking mode, whereby it is polled periodically to see if a choice has been selected.

■ **Syntax:**

```
CALL PullDnMSG(Choice$(), Status%(), Menu%, Choice%, Ky$, Action%)
```

■ **Where:**

Choice$() is a two-dimensional array containing the list of choices for each menu. If any element contains a hyphen only ("-"), it will be replaced by a separating line and will not be selectable by the user.

Status%() is a parallel, two-dimensional array that indicates which choices are active. Choices can be deactivated by assigning a non-zero value to the element that corresponds to a given item in the Choice$() array. The position of the hotkey is stored in the high byte of the Status%() array. If the 2nd letter is to be the hot letter, then you would store the value 1 * 256 in its respective Status%() element. Note that you store a value one less than the letter's position.

Menu% indicates which menu was active when a choice was selected, and may also be pre-loaded to force a given menu to be displayed initially.

Choice% indicates which choice was selected, and may also be pre-loaded to force a given choice to be highlighted initially.

**Ky$** holds the last key that was pressed by the user. This is used to determine if the user pressed Esc or Enter to exit the menu system. For example, if Ky$ is equal to CHR$(13) then the user has pressed Enter. Then by looking at the Menu% and Choice% variables, the menu and menu item currently selected can be determined.

Action% tells PullDnMSG how it is being used. The possible values for the Action% parameter are discussed in the section entitled "Multi-Tasking Menus" in Chapter 1.

Comments:

All of the colors for this routine are defined in the GPDat%() array. Their values are initialized in the file GETVIDEO.BAS. The benefit of isolating the color definitions to a single file is that you may customize them to your own preferences, and they will then be reflected in all of the programs that use GETVIDEO.BAS.

There are two routines which work with PullDnMSG to provide a complete user interface. The BarPrintMSG routine allows you to place the menu bar on-screen before the user has even entered the menu system. The MenuKeyMSG routine interprets the user's keystrokes and filters out Alt-key combinations which can be used to start the menu system. The MenuKeyMSG routine otherwise returns the keystroke that was pressed so that your program can interpret it. The MenuKeyMSG routine should replace the main INKEY$ statement in your program that is waiting for the users input. The program example DEMOPLMS.BAS shows a the minimum menuing setup for PullDnMSG.

■ **See Also:**

PullDownG

RandomFade

BASIC subroutine
contained in FADE.BAS

■ **Purpose:**

RandomFade randomly fades in portions of the second video page to the currently visible video page. This routine uses the FadeEGA routine to accomplish the effect of fading in the image.

■ **Syntax:**

```
CALL RandomFade
```

Comments:

It is advisable to use this routine and this effect when the overall background of the two screen images is going to be similar. Any time when there is a large portion of the screen which will not be changed by this routine, the effect is much more pleasing to the eye and doesn't appear rigid. See the demo program DEMOFADE.BAS for a visual description of this routine's functioning.

SetGPFont

BASIC subroutine
contained in DRAWTEXT.BAS

■ **Purpose:**

SetGPFont is designed to change between loaded fonts. Each font has separate information about its width and height. Even though these values may be the same, it's always advisable to use the proper information for the font. This routine sets up all of this information and requires only one step.

■ **Syntax:**

```
CALL SetGPFont(DesiredFontNum%)
```

■ **Where:**

DesiredFontNum% is the font number you wish to change to. This value will be honored, provided that the number is within the range of available fonts.

Comments:

There is only one time when you are required to call this routine. That is just prior to loading the font using the LoadFont routine. This is shown in the "Standard Code" section of Chapter 1.

■ **See Also:**

SetGWFont

SetGPSpacing

BASIC subroutine
contained in OUTLTEXT.BAS

■ **Purpose:**

SetGPSpacing is used to set the spacing for the GraphPak fonts to be drawn with either DrawText or StepText.

■ **Syntax:**

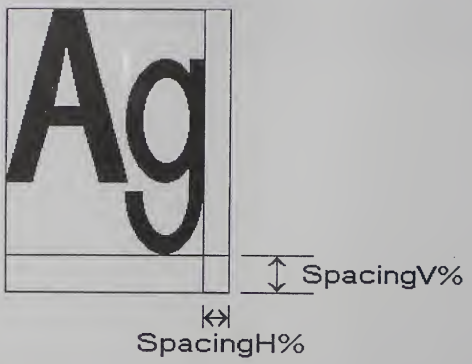
```
CALL SetGPSpacing(SpacingH%, SpacingV%)
```

■ **Where:**

SpacingH% sets the spacing between characters. This value is measured in pixels for the base font size.

SpacingV% sets the spacing between lines. This value starts at the position of the descender of a lower case letter such as 'g', and counts down in pixels.

Comments:



■ **See Also:**

SetGWSpacing

SetGWFont

BASIC subroutine
contained in OUTLTEXT.BAS

■ **Purpose:**

SetGWFont is designed to change between loaded Graphics Workshop fonts. Each font has separate information about its width and height. Even though these values may be the same, it's always advisable to use the proper information for the font. This routine sets up all of this information and requires only one step.

■ **Syntax:**

```
CALL SetGWFont(DesiredFontNum%)
```

■ **Where:**

DesiredFontNum% is the font number you wish to change to. This value will be honored, provided that the number is within the range of available fonts.

Comment:

There is only one time when you are required to call this routine. That is just prior to loading the font using the LoadFont routine. This is shown in the "Standard Code" section of Chapter 1.

■ **See Also:**

SetGPFont

SetGWSpacing

BASIC subroutine
contained in OUTLTEXT.BAS

- **Purpose:**

  SetGWSpacing is used to set the spacing for the Graphics Workshop fonts drawn by the OutlineText routine.

- **Syntax:**

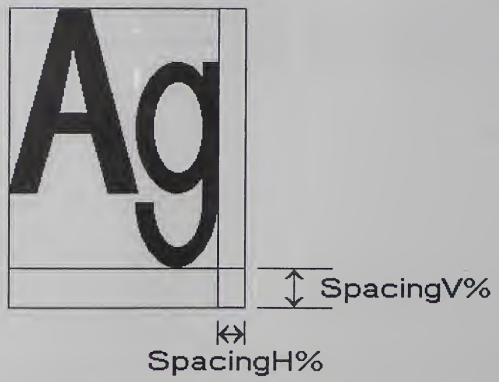
```
CALL SetGWSpacing(SpacingH%, SpacingV%)
```

- **Where:**

  SpacingH% sets the spacing between characters. This value is measured in pixels for the base font size.

  SpacingV% sets the spacing between lines. This value starts at the position of the descender of a lower case letter such as 'g', and counts down in pixels.

Comments:



- **See Also:**

  SetGPSpacing

SetVideo

BASIC subroutine
contained in SETVIDEO.BAS

■ **Purpose:**

SetVideo sets the screen mode and important screen information variables. This routine replaces the BASIC SCREEN statement as it allows you to select the best possible screen mode available with your monitor setup.

■ **Syntax:**

```
CALL SetVideo
```

Comments:

Add this routine as the last line of the standard code shown in Chapter 1. The code in GETVIDEO.BAS, or similar code which will determine the screen mode to be used, needs to be executed prior to calling SetVideo.

SetVideo uses the BASIC SCREEN statement even though Graphics Workshop has replacement routines which set the same screen modes. If you want to use any of the graphics routines like PAINT which have not been duplicated by routines in Graphics Workshop, you will need the BASIC SCREEN statement in order to use the BASIC PAINT statement. If you are designing a program that does not need any of the BASIC graphics statements, you can modify a version of SetVideo to use the Graphics Workshop routines. An example of this is the demo program VIEWPCX.BAS which creates a very small VIEWPCX.EXE by not using the BASIC graphics libraries.

SetVideo expects that the GPDat%() array exists and that element 31 contains the screen mode desired. See Appendix C for a description of element 31 of the GPDat%() array. The code in GETVIDEO.BAS sets up the GPDat%() array. Note that including the standard code described in Chapter 1 will fulfill these requirements.

Not only does the SetVideo routine set the graphics screen mode using the appropriate SCREEN statement, but it also sets other GPDat%() variables which can be helpful when programming for graphics mode. It sets elements 43 and 44 which hold the current pixel resolution of the screen. It sets element 49 which holds the aspect ratio for the current screen mode. It sets element 50 which holds the number of colors available in the current screen mode. It sets element 71 which holds the current character height of text drawn with the BASIC PRINT statement or any of the GPrint0VE routines.

ShadeH

BASIC subroutine
contained in SHADEH.BAS

■ **Purpose:**

ShadeH shades a region of the screen with a gradually changing bit pattern from one color to another.

■ **Syntax:**

```
CALL ShadeH(ULCol, ULLine, LRCol, LRLine, NumColors%, StepChoice%,_
  Colors%())
```

■ **Where:**

ULCol% and **ULLine%** specify the upper-left corner of the region to shade. **LRCol%** and **LRLine%** specify the lower-right corner of the region to shade. Together they specify the entire region to be shaded. These coordinates follow the mixed coordinate system where column values range from 1 to 80, and line values range from 0 to 479 on a VGA monitor.

NumColors% tells the routine how many color changes to go through.

StepChoice% tells the routine what the pattern will look like. The patterns are built almost randomly and use this value as the seed.

Colors%() is an array of the colors for the routine.

Comments:

This routine will start with Colors%(0) element as the background color, and the Colors%(1) element as the foreground color which will become more solid as the routine progresses. Once the color on screen becomes solid, Colors%(1) will become the background color and Colors%(2) will become the foreground color. The minimum dimension for the Colors%() array should be one greater than the value of the parameter NumColors%; the minimum value for the NumColors% parameter is 1.

The routine ShadeH is a faster version of the ShadeHorizontal routine. It is faster because it uses the mixed coordinate system. However the ShadeHorizontal routine allows you to use any pixel locations.

■ **See Also:**

ShadeHorizontal, ShadeV

ShadeHorizontal

BASIC subroutine
contained in SHADEH.BAS

■ **Purpose:**

ShadeHorizontal shades a region of the screen with a gradually changing bit pattern from one color to another.

■ **Syntax:**

```
CALL ShadeHorizontal(GWWindow AS Window1, NumColors%, StepChoice%,_
  Colors%())
```

■ **Where:**

GWWindow is a BASIC TYPE structure. The structure **Window1** is defined in the include file GWDECL.BAS. The variables in this record structure are **X1%**, **Y1%**, **X2%**, and **Y2%**. An example of programming with variables of this kind is shown at the beginning of this chapter.

NumColors% tells the routine how many color changes to go through.

StepChoice% tells the routine what the pattern will look like. The patterns are built almost randomly and use this value as the seed.

Colors%() is an array of the colors for the routine.

Comments:

This routine will start with Colors%(0) element as the background color, and the Colors%(1) element as the foreground color which will become more solid as the routine progresses. Once the color on screen becomes solid, Colors%(1) will become the background color and Colors%(2) will become the foreground color. The minimum dimension for the Colors%() array should be one greater than the value of the parameter NumColors%; the minimum value for the NumColors% parameter is 1.

The routine ShadeH is a faster version of the ShadeHorizontal routine. It is faster because it uses the mixed coordinate system. However the ShadeHorizontal routine allows you to use any pixel locations.

■ **See Also:**

ShadeH, ShadeVertical

ShadeV

BASIC subroutine
contained in SHADEV.BAS

■ **Purpose:**

ShadeV shades a region of the screen with a gradually changing bit pattern from one color to another.

■ **Syntax:**

```
CALL ShadeV(ULCol, ULLine, LRCol, LRLine, NumColors%, StepChoice%, _
  Colors%())
```

■ **Where:**

ULCol% and **ULLine%** specify the upper-left corner of the region to shade. **LRCol%** and **LRLine%** specify the lower-right corner of the region to shade. Together they specify the entire region to be shaded. These coordinates follow the mixed coordinate system where column values range from 1 to 80, and line values range from 0 to 479 on a VGA monitor.

NumColors% tells the routine how many color changes to go through.

StepChoice% tells the routine what the pattern will look like. The patterns are built almost randomly and use this value as the seed.

Colors%() is an array of the colors for the routine.

Comments:

This routine will start with Colors%(0) element as the background color, and the Colors%(1) element as the foreground color which will become more solid as the routine progresses. Once the color on screen becomes solid, Colors%(1) will become the background color and Colors%(2) will become the foreground color. The minimum dimension for the Colors%() array should be one greater than the value of the parameter NumColors%; the minimum value for the NumColors% parameter is 1.

The routine ShadeV is a faster version of the ShadeVertical routine. It is faster because it uses the mixed coordinate system. However the ShadeVertical routine allows you to use any pixel locations.

■ **See Also:**

ShadeH, ShadeVertical

ShadeVertical

BASIC subroutine
contained in SHADEV.BAS

- **Purpose:**

  ShadeVertical shades a region of the screen with a gradually changing bit pattern from one color to another.

- **Syntax:**

```
CALL ShadeVertical(GWWindow AS Window1, NumColors%, StepChoice%,_
  Colors%())
```

- **Where:**

  GWWindow is a BASIC TYPE structure. The structure **Window1** is defined in the include file GWDECL.BAS. The variables in this record structure are **X1%**, **Y1%**, **X2%**, and **Y2%**. An example of programming with variables of this kind is shown at the beginning of this chapter.

  NumColors% tells the routine how many color changes to go through.

  StepChoice% tells the routine what the pattern will look like. The patterns are built almost randomly and use this value as the seed.

  Colors%() is an array of the colors for the routine.

- **Comments:**

  This routine will start with Colors%(0) element as the background color, and the Colors%(1) element as the foreground color which will become more solid as the routine progresses. Once the color on screen becomes solid, Colors%(1) will become the background color and Colors%(2) will become the foreground color. The minimum dimension for the Colors%() array should be one greater than the value of the parameter NumColors%; the minimum value for the NumColors% parameter is 1.

  The routine ShadeV is a faster version of the ShadeVertical routine. It is faster because it uses the mixed coordinate system. However the ShadeVertical routine allows you to use any pixel locations.

- **See Also:**

  ShadeHorizontal, ShadeV

StepText

BASIC subroutine
contained in DRAWTEXT.BAS

- **Purpose:**

  StepText is used to draw a text string using the fonts available with GraphPak Professional.

- **Syntax:**

```
CALL StepText(X%, Y%, Text$, Angle%, Colr%, SizeMultiplier#)
```

- **Where:**

  The variables **X%** and **Y%** are pixel positions of the upper-left corner of the first character of the string to be drawn.

  Text$ is a string of the phrase to be drawn.

  Angle% is the angle in degrees at which the phrase will step. An angle of 0 draws text straight across the screen. All individual letters will be placed right side up.

  Colr% is the color of the text. Both DrawText and StepText phrases can have shadows drawn underneath them. Adding 128 to the color will activate the shadow effect.

  SizeMultiplier# is the size of the font. For example, 1# = same size as the original definition of the font, 2# = twice as big, .75 = 3/4 the size of the font.

Comments:

The color for the shadow effect mentioned above can be changed by modifying the variable GPDat%(14). See Appendix C for information about the GPDat%() array.

- **See Also:**

  GetTextWidth%, DrawText

VertMenuG

BASIC subprogram
contained in VERTMENU.BAS

Purpose:

VertMenuG is a comprehensive menu subprogram for graphics mode with many important capabilities including full support for a mouse. It always saves the underlying screen. Further, VertMenuG may be operated in a unique multi-tasking mode whereby it may be polled periodically to see if a selection has been made.

Syntax:

```
CALL VertMenuG(Item$(), Choice%, MaxLen%, BoxBot%, Ky$, _
  Action%)
```

Where:

Items$() is a conventional (not fixed-length) string array containing the list of menu choices.

Choice% indicates which choice was selected, and may also be pre-loaded to force a given choice to be highlighted when the menu system is accessed initially.

MaxLen% is the maximum length of any menu choice, thus establishing the menu width. Choices that are longer than MaxLen% will be displayed truncated.

BoxBot% is the bottom screen line that the window is to extend to. That is, if BoxBot% is set to twenty, then the bottom border of the menu will be on line twenty. Notice that the upper-left corner of the menu is established by the current cursor location.

Ky$ holds the last key that was pressed by the user.

Action% tells VertMenuG how it is being used. The different possible values are described in the section entitled "Multi-Tasking Menus" in Chapter 1.

Comments:

All of the colors for this routine are defined in the GPDat%() array. Their values are all initialized in the file GETVIDEO.BAS. The benefit of isolating the color definitions to a single file is that you may customize them to your own preferences, and they will then be reflected in all of the programs that use GETVIDEO.BAS.

VertMenuG is explained in depth in the section entitled "Multi-Tasking Menus" in Chapter 1, and a complete demonstration is provided in the DEMOVERT.BAS example program. This routine is also used by the DEMOMENU.BAS example program.

WhichPCXScreen

BASIC subroutine
contained in PCXHEADR.BAS

- **Purpose:**

  WhichPCXScreen interprets a PCX file header and determines the screen mode that should be used.

- **Syntax:**

```
CALL WhichPCXScreen(Array$, WhichScreen%)
```

- **Where:**

  **Array$** contains the header information for the .PCX file. Array$ is set by calling the routine OpenPCXFile%.

  WhichScreen% returns the suggested screen mode for the .PCX file. The value of WhichScreen% follows the values of the GPDat%(31) variable which is explained in Appendix C.

Comments:

This routine evaluates information in the header and determines the desired screen mode from this information. Even though not all of the information calculated in this routine is not used by the routine to determine the screen mode, it is done to give a complete breakdown of the PCX file header.

An example of using this routine is contained in the VIEWPCX.BAS example program, and the BASIC routine DisplayPCXFile.

OTHER FILES ON THE DISK

FILENAME	DESCRIPTION
HELV12.GFN	12 pt Helvetica GraphPak Font
HELV8.GFN	8 pt Helvetica GraphPak Font
OLDENG.GFN	12 pt Old English GraphPak Font
TROM12.GFN	12 pt Times Roman GraphPak Font
FUTURE.GFN	12 pt Future GraphPak Font
HELV12.GPK	GraphPak Font Definition
HELV8.GPK	GraphPak Font Definition
OLDENG.GPK	GraphPak Font Definition
TROM12.GPK	GraphPak Font Definition
FUTURE.GPK	GraphPak Font Definition
HELV.QFN	64 pt Helvetica GW Vector Font
QSEGUE.IN	QuickSegue Input File
GW.LIB	Graphics Workshop Library
GW.QLB	Graphics Workshop Quick Library
GW7.LIB	Graphics Workshop Library for BC7
GW7.QLB	Graphics Workshop Quick Library for BC7
DEMOSHAD.PCX	EGA .PCX file for Demonstration
FONTEDIT.PCX	EGA .PCX file for Demonstration
FONTS1.PCX	EGA .PCX file for Demonstration
FONTS2.PCX	EGA .PCX file for Demonstration
FONTS3.PCX	EGA .PCX file for Demonstration
XORBOX.PCX	EGA .PCX file for Demonstration
ZOOMED.PCX	EGA .PCX file for Demonstration
CGA.PCX	CGA .PCX file for Demonstration
MOUNTAIN.PCX	EGA .PCX file for Demonstration
MOUSE.PCX	HERC .PCX file for Demonstration
SALES1.PCX	EGA .PCX file for Demonstration
SALES2.PCX	EGA .PCX file for Demonstration
COMMON.GW	File of COMMON's for Graphics Workshop
GWDECL.BAS	DECLARES for Graphics Workshop
FADEDATA.GW	Data File for the FadeEGA Routine
GPFONT.GW	Initialization for GraphPak Fonts
EXTERNAL.GW	List of EXTRN's for Graphics Workshop
GWFONT.GW	Initialization for GW Vector Fonts
BC7START.GW	Assembler Include file for BC7 Compatibility
BC7END.GW	Assembler Include file for BC7 Compatibility

Chapter 5: QuickSegue

QuickSegue is a pr
transfer of .PCX files
a script input file an
read the script file an
You have the ability
screen in many differ
with text. QuickSegu
to be executed at an

About The Script La

A script used by Qui
can create an ASCII
on the Files menu in
the script file it cannot
comment lines in the

The statements in the

CLEAR
DO
GRAPHICS
LOAD
LOCATE
SOUND
SOUND
PAUSE
PRINT

Details About The Sc

CLEAR buffer

- **Purpose:**
  Clears a graphics memo
- **Where:**
  buffer is the number of
  to <type>
- **Purpose:**
  Allows programmers to
  for execution to add to a

QuickSegue is a programmable slide show program which combines transfer of .PCX files with simple graphics routines. QuickSegue accepts a script input file in a special format described below. QuickSegue will read the script file and manipulate the screen according to its statements. You have the ability to load .PCX files and bring them forward to the screen in many different fashions. You can also annotate your .PCX files with text. QuickSegue also gives you the ability to add your own routines to be executed at an appropriate time within the QuickSegue script.

About The Script Language

A script used by QuickSegue file can be any plain ASCII text file. You can create an ASCII text file by loading a document using the Load option on the Files menu in QuickBASIC. QuickSegue will ignore any lines in the script file it cannot understand, so you may place blank lines, or even comment lines in the script file.

The statements in the QuickSegue script language are:

```
CLEAR
DO
INTERLUDE
LOAD
LOCATE
SEGUE
SEND
PAUSE
PRINT
```

Details About The Script Language

CLEAR buffer

- **Purpose:**

  Clears a graphics memory buffer which has been previously loaded.

- **Where:**

  buffer is the number of buffers to clear.

```
DO dotype
```

- **Purpose:**

  Allows programmers to add their own sections of code to QSEGUE.BAS for execution to add to a presentation.

- **Where:**

  dotype is a number which is used to select one of the pre-programmed sections to execute. Currently only dotype = 1 is defined, and it moves an exclusive-OR box across the screen.

INTERLUDE number "anystring"

- **Purpose:**

  Starts one of the interlude types. Two pre-written interludes exist.

- **Where:**

  number is the number of the interlude. Number = 1 brings up a Movie Director's Clicker, using the phrase specified by **anystring**.

LOAD "filename" buffer

- **Purpose:**

  Loads a .PCX graphics file into a memory buffer.

- **Where:**

  filename is any valid filename including paths and extensions. If no path is specified, QSEGUE.BAS will assume the current directory.

  buffer is the number of one of the available buffers for loading the .PCX graphics file into.

LOCATE horizontal vertical

- **Purpose:**

  Locates an internal cursor for the positioning of text strings. This routine uses words to describe the position of a string since at the time of the locate command the size of the text string to be drawn is unknown.

- **Where:**

  horizontal is an X axis positioning command. Available horizontal positioning statements are "Left", "Middle", "Right".

  vertical is a Y axis positioning command. Available vertical positioning statements are "Top", "Middle", "Bottom".

SEGUE seguetype subtype color delay

- **Purpose:**

  Transfers a graphics screen from the background screen to the visible screen in one of many geometric fashions.

- **Where:**

  seguetype specifies the type of geometric transition to be made.

  subtype specifies the subtype of the above seguetype. Not all segue types have subtypes.

  color specifies the color to paint with for those subtypes which bring in a solid colored screen instead of a .PCX image from the background screen.

  delay specifies a time delay which can be used to uniformly control the speed of the transitions. The time is independent of the speed of the machine using the QSEGUE program and is measured in milliseconds.

SEND buffer

- **Purpose:**

  Sends a compressed graphics file from a buffer to the background video screen. Nothing is physically seen by the user when this command executes. It should be followed by a SEGUE statement in the script to bring the PCX image to the screen.

- **Where:**

  buffer is the number of one of the buffers which contains the .PCX graphics file to be displayed.

PAUSE

- **Purpose:**

  Waits for the user to press a keystroke.

```
PRINT "anystring" color
```

- **Purpose:**

  Prints a text string at the specified location in a helvetica font.

- **Where:**

  anystring is any string enclosed by quotes.

  color is the color for drawing the text string and is a number between 0 and 15.

Segue Types

All of the segue types described here are also described in Chapter 4, using pictures to better explain the process by which the segue takes place.

Segue 1 corresponds to a quick full screen transfer of the .PCX file.

Segue 2 corresponds to a inwardly imploding box transfer of the .PCX file.

Segue 3 corresponds to a random square replacement of the old screen with parts of the new .PCX file. The subtypes define the pattern of the replacement.

Segue 4 corresponds to a diagonal fade from the upper-left corner to the lower-right corner using parts of the .PCX file. The subtypes define the size of the blocks used to make up the line.

Segue 5 corresponds to a horizontal line replacement of the old screen with the new .PCX file. The subtypes define the number of lines being replaced at a time.

Segue 6 corresponds to a outwardly exploding box transfer of the .PCX file.

Chapter 6: Vector Fonts

Vector Fonts

A Vector Font is a s
font. Each letter c
Vector Font is drawn
filled in with the app

Because the Vector
necessary to underst
and the problems wh
when drawing vector
underneath where th
example in the Quick

```
SCREEN 9
LINE (75, 150) - (
LINE (75, 75) - (1
PAINT (60, 60), 2,
```

Because the fine cross
the border of the box
the box. Now consid
is any already existing
in the above PAINT s
both the line and the
command does not
prematurely and
selection is imp
using the vector
in the underlying ma
that your text is drawn
image with text, and t
image, the text will be

A Vector Font is a series of line segments which make up the outline of a font. Each letter can have its own number of line segments. When a Vector Font is drawn, its outline is generated first. The outline font is then filled in with the appropriate color.

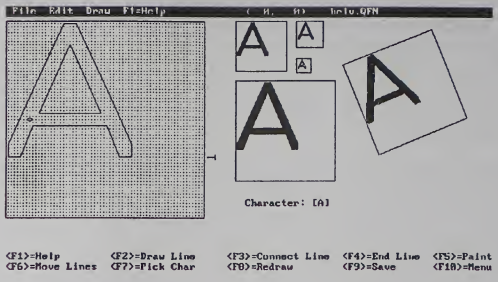
Because the Vector Font is dependent upon the PAINT command, it is necessary to understand a little about how the PAINT command works, and the problems which can occur while using it. A problem can occur when drawing vector fonts on a screen which already has graphics images underneath where the new text string will be drawn. Try the following example in the QuickBASIC editor:

```
SCREEN 9                                'turn on the graphics screen
LINE (75, 150) - (150, 75), 1           'assume previously on the screen
LINE (75, 75) - (125, 125), 1, B        'draw box to be painted
PAINT (80, 80), 2, 1                    'paint the box
```

Because the line crosses through the box and the line is the same color as the border of the box, the PAINT statement is unable to completely fill the box. Now consider that the box is the outline of a font, and the line is any already existing graphic image on the screen. Note: The first color in the above PAINT statement is irrelevant. The problem occurs because both the line and the box are drawn in the same color. The PAINT command does not know that the two are separate and therefore stops prematurely and does not paint the entire box. For this reason, color selection is important when drawing a Vector Font. If you are going to be using the vector fonts over already existing images, use a color not used in the underlying image. This is good for two reasons. One, it will insure that your text is drawn completely. Two, when you annotate a graphic image with text, and the text uses a different color than parts of your image, the text will be easily discernable from the background image.

Using the Vector Font Editor

The Vector Font Editor is called FONT64.EXE and is a program for editing a font definition. The grid used in FONT64.EXE gives fonts up to a 64 x 64 point resolution. The Vector Font editor shows what the font will look like for 4 different sizes. One view of the font definition shows how the character will look at an angle.



Start the Vector Font Editor by typing:

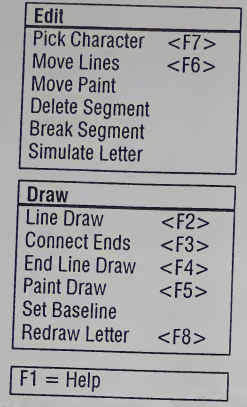
```
FONT64
```

Once the Vector Font Editor has started, there are a couple of basic commands which can be used to create and edit vector fonts. The first of these is the <F10> key which is used to activate the menu system. The second is the <F1> key which is used to activate FONT64's help system.

Using the Menu System

The menu system is activated by pressing the <F10> key. If a mouse is available, selecting on any of the menu items at the top of the screen will activate the menu system. This section describes each of the options on each of the menus in the Vector Font Editor.

Files	
New Font	
Open Font...	
Save Font...	<F9>
Save Font As...	
Load Simulate Font...	
DOS Shell	
Exit	



Detailed Function Description Of Menu Items

Items on the Files menu

- **New Font**
  This option clears the memory of all letters of a font. The user is prompted if any changes have not yet been saved.

- **Open Font...**
  This option is used to load a predefined font file for the purpose of editing. The user is prompted to save the current file if any changes have been made since the last save. The user will also be prompted for a filename of the font file. All vector font files have the .QFN extension, by default, so this need not be typed.

- **Save Font...**
  This option is used to save the current font information to the filename specified by a previous open command. If no previous open has been executed then this functions the same as the Save As command and the user will be prompted for the filename to save the font under. Otherwise no further prompt will be given to the user before saving the font file.

- **Save Font As...**
  This option is used to give a font a new name, or to name a font created with the New Font command. The user is prompted for the name to be given to the font file. All vector font files have the .QFN extension.

- **Load Simulate Font...**
  This option is used to load a simulation font from one of the fonts provided from GraphPak. GraphPak fonts have the .GFN extension, and a list of available GraphPak font files is in chapter 4. FONT64 will default and use the Helv12.GFN font file.
- **DOS Shell**
  This option is used to shell to DOS to utilize some of its functions.
- **Exit**
  This option exits the font editor. If your file has not been saved, you will be prompted to save it.

Items on the Edit menu

- **Pick Character**
  This option is used to determine which character's font description to edit.
- **Move Lines**
  This option is used to move the endpoint of two connected lines. Two line segments are connected at one point and the user must place the square cursor over the connecting point of these two lines. After selecting Move Lines, moving the cursor will also move the connecting point of these two lines and the lines will change on screen. Pressing <Enter> will finalize the move.
- **Move Paint**
  This option is used to move the center point for the painting region. The user must have the cursor over one of the paint points before using this command. Pressing <Enter> will finalize the move.
- **Delete Segment**
  This option will remove one of the line segments from the font description. This option will start by changing the color of one of the line segments which make up the font definition. You select which segment to delete by pressing the <SpaceBar> until that segment is the one with the different color. Once the proper segment is selected, press <Enter>. You will be prompted to verify the removal of that segment.
- **Break Segment**
  This option is used to insert line segments into the font structure at the specified position. This option will start by changing the color of one of the line segments which makes up the font definition. You select which segment to break into two segments by pressing the <SpaceBar> until that segment is the one with the different color. Once the proper segment is selected, press <Enter>. At this point the line segment will be broken into two and the cursor positioned halfway between the two endpoints of

the first segment. Now the cursor keys can be used to position the midpoint of the two line segments. To finalize the position of the midpoint for the two line segments, press the <Enter> key.

Items on the Draw Menu

- **Line Draw**
  This option begins the process of drawing a line segment. The line segment begins at the current cursor position and ends where the cursor is when either the End Line Draw command or the Connect Ends command is issued.

- **Draw Another**
  This option replaces the Line Draw option when a line draw is in progress. This option will draw a line segment from the last cursor position where a Line Draw or Draw Another command was issued to the current cursor position. The routine is also prepared to receive additional Draw Another commands to draw a line from the current cursor position to wherever the cursor position is when those commands are issued.

- **Connect Ends**
  This option draws a line between the cursor position that existed when the original Line Draw (not Draw Another) command was issued to the current cursor position. At the same time, this option ends the line drawing mode. It will restore the Line Draw option to the Draw menu. The next Line Draw command will create a line segment which is not connected in any way to the line segment just drawn.

- **End Line Draw**
  This option ends line drawing mode. It will restore the Line Draw option to the Draw menu. The next Line Draw command will create a line segment which is not connected in any way to the line segment just drawn.

- **Paint Draw**
  This option places a paint point marker at the position where the PAINT command is to begin to fill the outlined object. Multiple paint points can be placed in a character. A section of a font may be closed off from another section resulting in an incomplete painting of the entire character. When displayed at smaller sizes, this can happen near curves and edges. The solution is to place more than one paint point in a letter's font definition.

- **Set BaseLine**
  This option sets the baseline for the font to be at the cursor's current vertical position. The baseline is the position which is used to align letters with and without descenders. A descender is the portion of a lower case letter such as 'g' which extends below most other letters. This baseline

can be used to determine the height of a vector font letter. To determine the height of a font, see Interlude1 and its use of the 2nd byte in the font description.

- **Redraw Letter**
  This option redraws the font definition for the currently selected character on screen. If a font definition is not made of completely enclosed objects, the PAINT command could paint outside the boundaries of the region. If this happens, using this command will clean up the screen.

Items on the Help Menu

The Help Menu has no items. There are no help options; there is only one method of obtaining help. Simply press <Enter> on this menu to activate help.

Using Vector Fonts With Your Program

The routine that draws a vector font is called OutlineText. This routine accepts a physical screen location to draw the string of text and two variables to use for sizing the font. These variables are integer values which correspond to the numerator and the divisor of some fraction which will be multiplied by the font definition. Basic can generate integer multiplies and divides which will be much faster than using floating point arithmetic. Almost any size font can be generated. An algorithm for obtaining an integer numerator and denominator for some real number contained in A! is:

```
Numerator% = A! * 1000
Denominator% = 1000
```

The algorithm for drawing a string works just as fast with the Denominator equal to 1000 as when it is equal to 1. Using a Denominator of 1000 provides a wide range of font sizes.

Appendices

Appendices

Appendix A: A PC

For some time now, t
graphics, and if you
formats. The PCX g
PC Paintbrush and
program can read a
the PCX format wa
until this change, a
designed for the 256

To start, like many d
used to describe the
we can do anything w
bytes in length. Note
would have been no n
other VGA modes.

Header portion

The first byte of the
defines the file as a PC
this fact that the
PC Paintbrush fi

The second byte is
file format it is. The
this is whether or not
palette. The version
following table:

VERSION	DESCRIPTION
0	Ver
2	Co
3	Co
5	No

PC PaintBrush IV uses

The third byte is the
the value 1 to represent

The fourth byte is the num
used in most of the EGA

Appendix A: A PCX PRIMER

For some time now, the three letters PCX have come to mean bit mapped graphics, and if you want to display bit mapped graphics you use the PCX format. The PCX graphics format was created by ZSoft, the makers of PC Paintbrush and Publishers PaintBrush. Just about every graphics program can read a .PCX file and display it in some fashion. Until 1989, the .PCX format was left unchanged. We'll cover the .PCX format up until this change, and then discuss the addition which basically was designed for the 256 color mode of the VGA.

To start, like many data files, the .PCX file has a header portion which is used to describe the image. We will need to examine this header before we can do anything with the image. The size of the header is always 128 bytes in length. Note: If the header had been made variable in length there would have been no need to modify the standard to accommodate the 256 color VGA modes.

Header portion

The first byte of the header always contains the value 10 (0A hex) which defines the file as a PC Paintbrush PCX file. Use this in combination with the fact that the file has the .PCX extension to confirm that the file is a PC Paintbrush file.

The second byte is the version number. It tells you which version of the file format it is. The important information which can be derived from this is whether or not the file contains information about a re-mapped palette. The version numbers which ZSoft has published are in the following table:

VERSION	DESCRIPTION OF FORMAT
0	No Palette Information
2	Contains Palette
3	No Palette Information
5	Contains Palette

PC PaintBrush IV uses version 5 of the .PCX format.

The third byte is the compression scheme used. This byte will always have the value 1 to represent "run-length encoding", which is discussed later.

The fourth byte is the number of bits per pixel. This ignores color planes used in most of the EGA and VGA modes. Most of the time this byte will

be 1. The exceptions are the CGA 4-color mode where this value is 2, and the VGA 256-color mode where this value is 8.

The fifth through twelfth bytes store the picture dimensions as 4 integer values. The dimensions are shown in the order XMinimum, YMinimum, XMaximum, and YMaximum.

The next four bytes (13 through 16) store two integer values of the physical resolution possible for the machine on which the image was created. If it was created on an EGA display, for example, these values would be 640 and 350.

The next 48 bytes (17 through 64) store the palette information. The palette is discussed in Appendix B.

Byte 65 in the file is reserved by ZSoft for future use.

Byte 66 in the file tells you the number of color planes used by the image. When using one of the 16 color modes on the EGA or VGA, the number of color planes is set to 4; otherwise it is set to 1.

Byte 67 is very important. It defines the number of bytes per line the image uses. A VGA screen which is 640 pixels across uses 80 bytes to store the information (ignoring multi-plane aspect). The value in this byte has always been recorded correctly. Some versions of paint programs have written incorrect values in the header bytes which describe the picture dimensions. One common mistake describes EGA screens as being only 75 by 75 pixels in dimension. However, even in the .PCX files with that incorrect picture dimension, the value in Byte 67 was still correct, 80 bytes.

The remaining 61 bytes in the header are unused.

Data Portion

The data portion contains compressed bytes which hold pixel values. Since each video mode has a different way of storing pixels on the screen, and the PCX file format stores information for each screen in a fashion that closely resembles the memory of the video screen the image was taken from, you cannot directly interpret the information in the data portion. It is necessary to use the header information given above to determine the monitor used, and then interpret the data portion accordingly.

For those unfamiliar with compression, it was invented to allow storage devices to hold more information. An EGA 16-color high resolution screen has 80 bytes per scan line, 350 scan lines per video plane, and four

video planes. That's 91,200 bytes of information. It would be nice to store that same information in less space. That's where data compression comes in. Put simply, if 40 bytes in a row are all the same, it is simpler to say the next 40 bytes are all the same and this is what the byte is. That takes only two bytes to represent: one to say that the next 40 bytes are the same, and the other to say what that byte is. Two bytes instead of 40 provides a compression of 95%. Unfortunately, the entire file doesn't usually compress that well; a typical average is around 60%.

The next problem is how to represent a count and how to represent a data byte. Since a data byte can have any of the possible values from 0 to 255, there are no special codes left to use for counts. The people at ZSoft decided to use the numbers from 0 to 191 for data and the numbers 193 to 255 for counts. The value of 192 is not used, for reasons described later. The simple way of looking at a byte to see if it is a count and not data is to look at the leftmost two bits of the byte.

11000000

If the leftmost two bits are set, then subtract the value 192 from that byte and the resulting value will be the count of how many times the following byte is going to be repeated. Note: this byte which follows can be any value from 0 to 255. If we had only a single byte with a data value larger than 192, then we would have to say here comes 1 byte and its value is X. In the cases where a single byte exists and it is larger than 192, it is stored as two bytes instead of one. This can be a problem in more complicated graphics, but it's rare for this compression scheme to use more memory than the image originally took up.

This compression continues for the remainder of the file. Now let's answer the remaining questions: How do I handle EGA video planes? Why isn't a data value of 192 used?

The EGA video planes are stored on a line-by-line basis. For each line, the blue plane's data is stored, followed by the green, red and intensity planes. Some programs will compress across the boundaries of a plane and even across a scan line. For example, if the last 5 bytes of a line are zeros, you would expect to get a repeat count of 5 and then a zero. But, if the first 3 bytes of the next line or even the next video plane are also zeros, what you actually get is a repeat count of 8 followed by a zero. This all-out compression makes a compressed file smaller as a rule, but raises the time it takes to display a .PCX file because of the extra checking for end of line and end of plane.

To see why a data value of 192 isn't used, remember the .PCX format rules: We must look at the two leftmost bits to see if they are both ones.

If they are, then subtract 192 from that byte to obtain a count. Both leftmost bits of 192 are ones. Since 192 - 192 is zero, and you can't have a count of zero, the value of 192 becomes meaningless and is not used in the .PCX format.

APPENDIX B: THE PALETTE

A painter often uses few colors to paint his masterpiece. If it is a water scene, you can be sure there will be some blue, some green and some white. With a painting using oil paints, the artist has the ability to blend these colors on the canvas. His whitecaps on the water can blend and make light blue with the water beneath. In simplest terms he has two colors in the same place at the same time. A computer screen can't have two colors for the same pixel, but we do have a palette. A palette is a selection of colors we can choose from to paint our picture.

On EGA and VGA screens there are 16 color graphics modes. On these screens there is a palette of 16 colors. You don't always need every color. In particular, for a water scene you need blue, dark green and white. You won't need a yellow, red, purple, violet, bright green or a pink. You've just eliminated the need for about half of the colors given to us in the standard palette. Luckily, the palette gives us the ability to replace any of the colors we won't need with another color. On the EGA you can select from 64 different colors; on the VGA you can select from 256,000. We'll go into more on how these colors are structured later, but for now let's say we can have 4 shades of blue, 4 shades of green, and 4 shades of white. That's twelve colors to paint with instead of three. That's obviously going to create a better picture than if only three colors were used. Look at the end of the demo program DEMOPAL.BAS; it shows the effect of painting with the right colors.

How The Palette Works

The EGA and VGA monitors are the same in many ways. However, one way they are completely different is in the number of colors available and in how these colors can be accessed.

How The EGA Stores Palettes

The EGA has a maximum of 16 colors which can be displayed at any one time. Each of these 16 colors can be among one of the 64 possible choices for the colors. To represent 64 colors requires a 6-bit binary number. Among these 6 bits are 2 bits for the Red value, 2 bits for the Green value, and 2 bits for the Blue value. One bit for each color is used as a high-intensity value. In the explanation below, the high-intensity bit is represented by the capital letter, and the low-intensity bit is represented by the lower-case letter. Each of the values for the RGB value stored here is represented in the following fashion: 00RGBrgb. Each pair: Rr, Gg, and Bb, has one of four values: 00, 01, 10, or 11. This means that if the

bits represented by Rr are both zero, then there is no red in the color generated. The same is true of the other colors. If you set any of its bits to 1, you are selecting the intensity of the color you are adding to the original color. The brightness of a color is determined by the overall number of bits which are set to a value of 1. The color 56 has only three bits which are 1's, while 55 has five bits which are 1's; thus the color 55 will be considerably brighter than the color 56.

Setting the EGA's palette is accomplished by sending a palette register number and a value to a routine which can set the palette. The routine SetPaletteEGA will do this for you. The EGA uses "write-only" registers, so it is not capable of telling you what are the values of the palette registers currently in use. You will need to maintain this information for yourself.

How The VGA Stores Palettes

The VGA can store palettes the same way, but it has the ability to improve the process by expanding the number of available colors. You can specify any of 64 values for each of the main colors red, green, and blue. This provides an overall color palette of 256,000 colors. Graphics Workshop has a routine called SetPalTripleVGA which has the number of the palette register and then three values for the Red, Green and Blue color values, respectively. The higher the overall values of each of the three color values, the brighter the color will be. The VGA 256-color mode allows any of its 256 colors to have one of the 256,000 colors mentioned above. Some day computers will be giving us more colors to choose from, but for now that's the maximum we can accommodate on a PC and that's the maximum which QuickBASIC can handle.

The palette is stored by the Video BIOS. When the computer starts up, it looks to the Video Card installed in your machine. The video card has encoded on it important information about the video card which is then referenced by the BIOS. This is made up of tables, code for initializing the video card into one of its possible video modes, and code for sending pixels of colors to the screen.

What Can You Use Palettes For?

Now that we have the code and knowledge for using palettes, what immediate uses do we have for it? In Graphics Workshop there are many routines for displaying PCX graphics files. These graphics files were probably made with PC Paintbrush or some other graphics program (possibly even with the Graphics Workshop SavePCXVE subprogram). These programs all have the ability internally to change the palette of the

image being displayed or created. To display these PCX files, use the native palette which was used to create it. Other uses discussed below are covered in the demonstration program DEMOPAL.BAS. Another use is drawing video data to the screen and then recoloring it without having to redraw it to the screen.

One could also use palettes for the creation of a three dimensional video game. The terrain in this demo can have complex images (the lines) drawn over them and it would appear that the terrain is being redrawn each time you want to move it. Obviously, redrawing the background image under these complex images would be very complicated and slow. Instead, the color of individual places on the screen is modified, allowing the program to spend valuable CPU time more efficiently. Palettes are good for displaying motion without actually using effort to redraw the screen.

The Logo contained in the demo program makes use of a flowing palette. Each of the colors follows one another in a succession which can make the image appear to be a solid color.

These types of effects will make a presentation much more effective than is possible with just one color.

Appendices

APPENDIX C: THE GPDAT%() ARRAY FROM GRAPHPAK

GraphPak Professional is another product produced by Crescent Software. It is a package for doing business charts and scientific graphs. You may be wondering why we talk about GraphPak Professional in this product. We recognize that a lot of GraphPak Professional owners are going to want to use some of the Graphics Workshop routines with GraphPak Professional, and since it's nice to have some of the features of the GraphPak Professional GPDat%() array in any program having to deal with graphics monitors, we've used it in Graphics Workshop. GraphPak's Professional GPDat%() array has elements which hold the current screen mode set, the maximum pixel resolution of the display, the maximum number of colors on the display, the color values for pull-down and vertical menus, and vital values for using the GraphPak Professional fonts. GraphPak Professional fonts are those which are displayed using the DrawText and StepText routines, which we have also supplied in this product.

Description of the GPDat%() Array

The GPDat%() array was created to share information between routines. Inside a subroutine in QuickBASIC, you don't have access to all the variables in your main program. Usually you'll have access only to those variables which you have passed to the subroutine. Using the COMMON statement in QuickBASIC, you can reach variables from outside the subroutine. Variables that we want to be able to access from inside various routines are font definitions, and the GPDat%() array. The requisite common statements have all been included in the COMMON.GW module which should be included at the top of each of your modules. The GPDat%() array is an array of integers. It is dimensioned to 85 elements, of which the first 80 are used by GraphPak Professional routines. Not all of these are needed for the GraphPak Professional font routines included with this package, but some elements which are needed have significant uses, such as allowing you to specify boldfacing, italicizing, etc. The GPDat%() array is broken up into two sections, system variables and user variables. System variables hold important system parameters, like screen size. User variables are used to modify the routines like DrawText and StepText work with the font information given to them.

Many of the items in the GPDat%() array are integers, but some are used simply as Yes or No indications, or boolean variables. A boolean variable is one that has two states, either TRUE or FALSE. In computers a FALSE is represented by a 0, whereas TRUE is commonly represented by -1.

BASIC considers any non-zero value to be TRUE, but try always to use -1, as it is more of a standard.

Elements Used By Graphics Workshop

ELEMENT	VARIABLE DESCRIPTION	DATA (RANGE)
14	Text Shadow Color	Integer (0-15)
23	Boldface Text	Boolean (-1 or 0)
24	Italicize Text	Integer (45 to 135)
31	Monitor Type	Integer (0-9)
33	Maximum Fonts Available	Integer
34	Current Font Number	Integer (< = Fonts Avail)
35	Maximum Font Width	Integer
36	Font Vertical Spacing	Integer
43	Video Monitor Width	Integer (320, 640, or 720)
44	Video Monitor Height	Integer (200, 350, or 480)
47	Horizontal Text Spacing	Integer
48	Vertical Text Spacing	Integer
49	Screen Aspect Ratio	Integer (0 - 1000)
50	Maximum Colors	Available
57	Background Screen Color	Integer (0 to 15)
59	Temporary Text Spacing	Integer (-15 to 15)
71	Character Height	Integer (8, 14, 16)
72	Graphics Storage Segment	Integer (> &HA800)
73	Mouse Active	Boolean (-1 or 0)
74	Highlight Bar Color	Integer (GW Color)
75	Pull-down Box Color	Integer (GW color)
76	Active Item Color	Integer (GW color)
77	InActive Item Color	Integer (GW color)
78	Active Menu Color	Integer (GW color)
79	InActive Menu Color	Integer (GW color)
80	Normal Screen Color	Integer (GW color)
81	GW Fonts Available	Integer
82	GW Font Active	Integer
83	GW Horiz Font Spacing	Integer
84	GW Vert Font Spacing	Integer
85	GW Overall Font Height	Integer
86	Draw an Outline Only	Boolean (-1 or 0)

Note:

Variables without ranges can have any range available to an integer variable. A GW color refers to colors designed to be displayed with the GPrint0VE routine. The formula for a GW (Graphics Workshop) color is:

```
GWcolor = Foreground + 256 * Background
```

See the detail on the following pages for more information.

Elements 14, 23, 24, 33, 34, 35, 36, 47, 48 and 59 all have effects on the GraphPak Professional proportional font system.

Elements 31, 43, 44, 49, 50, 57, 71, 72 and 73 are all system variables which give vital information about the screen mode in use and if a mouse is present.

Elements 74 through 80 all are used by the graphics pull-down, vertical menus and message box routines to pass the colors of the individual parts of these structures.

Elements 81 through 86 all have effects on the Graphics Workshop Vector font system.

Detailed Listing Of Elements Used In Graphics Workshop

GPDat%(14) : Text Shadow Color

- **Data Type:**

  Integer

- **Possible Values:**

  Must be between 0 and 15. On some screen modes the range is less, but higher values will display as the highest available color for that screen mode (e.g. on CGA 4-color mode the range is 0 to 3. A value of 14 in this variable will be considered as a 3). GPDat%() element 50 has the maximum number of colors for the current screen mode.

- **Uses:**

  Text Shadowing can create interesting displays for major titles and presentations. This element affects only the GraphPak fonts.

- **Note:**

  This element is automatically initialized to zero by QuickBASIC.

GPDat%(23) : Boldface Text

- **Data Type:**

  Boolean

- **Possible Values:**

  -1 = Boldface the text
  0 = Normal text

- **Uses:**

  Boldfacing adds emphasis to points of interest on a graph. This element affects only the GraphPak fonts.

- **Note:**

  This element is automatically initialized to zero by QuickBASIC.

GPDat%(24) : Italicize Text

- **Data Type:**

  Integer

- **Possible Values:**

  This variable represents the angle for italicizing. Normal text uses an angle of 90 degrees. To italicize, use an angle of about 60 degrees. To obtain backwards italics use an angle above 90. A reasonable range for this variable is 45 to 135 degrees.

- **Uses:**

  Italicizing helps to emphasize points of interest on a graph. This element affects only the GraphPak fonts.

- **Note:**

  GETVIDEO.BAS initializes this variable to 90.

GPDat%(31) : Monitor Type Used

- **Data Type:**

  Integer

- **Possible Values:**

  0 = Monitor Unknown
  1 = EGA with Mono Monitor
  2 = Hercules Graphics Adaptor
  3 = Monochrome (not capable of graphics)
  4 = CGA Graphics 4 colors
  5 = EGA Graphics 16 colors
  6 = CGA Graphics 2 colors
  7 = Mono EGA Graphics
  8 = VGA Graphics 16 colors
  9 = VGA Graphics 256 colors

- **Uses:**

  This variable is used by most of the Graphics Workshop routines to create the appropriate positioning and scaling. It is also used by SetVideo to set the correct screen mode.

- **Note:**

  This variable is initialized by the function MultMonitor%, which is called within GETVIDEO.BAS.

GPDat%(33) : Maximum Fonts Available

- **Data Type:**

  Integer

- **Possible Values:**

  Must be at least 1 and can be as large as string memory will allow.

- **Uses:**

  When the Current Font Number is set using the routine SetGPFont, this variable is used to ensure that the font actually exists. The include file GPFONT.GW will dimension the Font$() array to 95 elements by GPDat%(33) elements in size. If you want to have more than one GraphPak Professional font, set the variable MaxGPFonts% to the desired number of fonts prior to the include file GPFONT.GW. This element affects only the GraphPak font system.

- **Note:**

  GETVIDEO.BAS initializes this variable to 1. It will be reset to the value of MaxGPFonts% if the variable/constant contains a non-zero value.

GPDat%(34) : Current Font Number

- **Data Type:**

  Integer

- **Possible Values:**

  Must be at least 1 and can be as large as the maximum number of fonts. (See GPDat%(33))

- **Uses:**

  Allows for different fonts to be displayed on the same screen. Proper usage of this feature can produce spectacular effects. This element affects only the GraphPak font system.

- **Note:**

  GETVIDEO.BAS initializes this variable to 1.

GPDat%(35) : Maximum Font Width

- **Data Type:**

  Integer

- **Possible Values:**

  Any value between 0 and 16 is possible. The value is determined by measuring the width of the capital letter 'W'. In most fonts this will be the widest letter.

- **Uses:**

  Allows StepText to display a centered label, while still using the proportional font system. This element affects only the GraphPak font system.

- **Note:**

  This variable is set by the routine SetGPFont.

GPDat%(36) : Font Vertical Spacing

- **Data Type:**

  Integer

- **Possible Values:**

  This variable is set to the combination of GPDat%() element 48 and the Font's Maximum Height as determined by the LoadFont routine. The Font's Maximum Height is contained in the FontHeight%() array. This value will typically be between 8 and 16.

- **Uses:**

  Many of the routines use this variable to determine spacing from one line to another. This element affects only the GraphPak font system.

- **Note:**

  This variable is set by the routine SetGPFont.

GPDat%(43) : Video Monitor Width

- **Data Type:**

  Integer

- **Possible Values:**

  Possible values are 320, 640 or 720, depending on the screen being used.

- **Uses:**

  Many routines use this value to determine centering, and to avoid writing outside the screen's boundaries.

- **Note:**

  This variable is set by the routine SetVideo, depending on the screen mode actually used.

GPDat%(44) : Video Monitor Height

- **Data Type:**

  Integer

- **Possible Values:**

  Possible values are 200, 350 or 480, depending on the screen being used.

- **Uses:**

  Many routines use this value to determine centering, and to avoid writing outside the screen's boundaries.

- **Note:**

  This variable is set by the routine SetVideo, depending on the screen mode actually used.

GPDat%(47) : Horizontal Text Spacing

- **Data Type:**

  Integer

- **Possible Values:**

  Any value from -15 to 15 can be used.

- **Uses:**

  This variable sets the number of pixels to appear as separators of any two letters printed on the screen. It is also the natural width for the space character. This element affects only the GraphPak font system.

- **Note:**

  This is set by the SetGPSpacing routine, but is initialized in GETVIDEO.BAS to 2 pixels.

GPDat%(48) : Vertical Text Spacing

- **Data Type:**

  Integer

- **Possible Values:**

  Any value from -15 to 15 can be used.

- **Uses:**

  This variable sets the number of pixels to move beyond the descender of any of the lower case letters like 'g', 'y', and 'j', before writing the next line of text. This element affects only the GraphPak font system.

- **Note:**

  This is set by the SetGPSpacing routine, but is initialized in GET-VIDEO.BAS to 2 pixels.

GPDat%(49) : Screen Aspect Ratio

- **Data Type:**

  Integer

- **Possible Values:**

  Any value between 0 and 1000. This value is determined by the following equation:

```
GPDat%(49) = INT((4 * (GPDat%(44) / GPDat%(43)) / 3) * 1000)
```

- **Uses:**

  This is used for drawing text at angles and drawing circles. The purpose of this value is to allow circles to appear perfectly circular, no matter what the screen resolution. It can also be used to adjust results of the SIN function when used for displaying rotated three-dimensional graphics images.

- **Note:**

  This value is set in the routine SetVideo, depending on the screen mode actually used.

GPDat%(50) : Maximum Colors Available

- **Data Type:**

  Integer

- **Possible Values:**

  Possible values are 2, 4, 16, or 256 depending on the screen activated in the routine SetVideo.

- **Uses:**

  This variable allows the internal routines to know how many colors they have to work with.

- **Note:**

  This value is set in the routine SetVideo.

GPDat%(57) : Background Screen Color

- **Data Type:**

  Integer

- **Possible Values:**

  Possible colors are 0 - 15.

- **Uses:**

  The routine GPaintBox uses this variable to determine the background color of the screen.

GPDat%(59) : Temporary Text Spacing

- **Data Type:**

  Integer

- **Possible Values:**

  Pixel offset from -15 to 15 for the base font size. To achieve a true pixel offset, all text routines automatically multiply this number by the size of the text.

- **Uses:**

  One use is in underlining text, since all proportional fonts have spaces after each character. You can set this variable to counteract the effects of GPDat%() System element 47. (Use this element rather than changing element 47. It is good practice not to change the System elements.) This element affects only the GraphPak font system.

- **Note:**

  This variable is initialized to 0. Reset this variable to 0 when you have finished using it.

GPDat%(71) : Character Height

- **Data Type:**

  Integer

- **Possible Values:**

  For the CGA displays the height of the character font is 8 pixels. For EGA displays the height is 14 pixels. For VGA displays the height is 16 pixels.

- **Uses:**

  The PullDownG and VertMenuG systems use this information for placement of graphics elements, and for conversion between coordinate systems. This value can also be used for placement of text when using the mixed system.

- **Note:**

  This variable is set by the routine SetVideo, depending on the screen mode actually used.

GPDat%(72) : Graphics Storage Segment

- **Data Type:**

  Integer

- **Possible Values:**

  Since this variable is used to store the next possible location to save VGA or EGA video memory, it is going to be some value above &HA800. The segment &HA800 is the starting location for the second EGA video page. The segment &HAA00 will be the starting location for the second VGA video page, if there is enough memory to have a complete second VGA video page.

- **Uses:**

  The routines PullDownG, VertMenuG, and MsgBoxG use this segment value to know where they can save their next information.

- **Note:**

  This variable is set inside the routines PullDownG, VertMenuG, and MsgBoxG.

GPDat%(73) : Mouse Active

- **Data Type:**

  Boolean

- **Possible Values:**

  -1 Mouse driver is loaded and mouse initialized
  0 Mouse not available

- **Uses:**

  The routines PullDownG and VertMenuG use this value to determine whether or not to execute mouse specific code.

- **Note:**

  This variable is set inside the include file GETVIDEO.BAS by calling the InitMouse routine.

GPDat%(74) : Highlight Bar Color

- **Data Type:**

  Integer

- **Possible Values:**

  Possible values include color values which are designed for the GPrint0VE routine. The formula for combining a foreground color and a background color is

```
GColor = Foreground + 256 * Background
```

- **Uses:**

  The routines PullDownG, VertMenuG, and MsgBoxG use this color to determine colors on the screen under the right circumstances.

- **Note:**

  This variable is set inside the include file GETVIDEO.BAS.

GPDat%(75) : Pull-down Box Color

- **Data Type:**

  Integer

- **Possible Values:**

  Possible values include color values which are designed for the GPrint0VE routine. The formula for combining a foreground color and a background color is

```
GColor = Foreground + 256 * Background
```

- **Uses:**

  The routines PullDownG, VertMenuG, and MsgBoxG use this color to determine colors on the screen under the right circumstances.

- **Note:**

  This variable is set inside the include file GETVIDEO.BAS.

GPDat%(76) : Active Item Color

- **Data Type:**

  Integer

- **Possible Values:**

  Possible values include color values which are designed for the GPrint0VE routine. The formula for combining a foreground color and a background color is

  ```
  GColor = Foreground + 256 * Background
  ```

- **Uses:**

  The routines PullDownG, VertMenuG, and MsgBoxG use this color to determine colors on the screen under the right circumstances.

- **Note:**

  This variable is set inside the include file GETVIDEO.BAS.

GPDat%(77) : InActive Item Color

- **Data Type:**

  Integer

- **Possible Values:**

  Possible values include color values which are designed for the GPrint0VE routine. The formula for combining a foreground color and a background color is

  ```
  GColor = Foreground + 256 * Background
  ```

- **Uses:**

  The routines PullDownG, VertMenuG, and MsgBoxG use this color to determine colors on the screen under the right circumstances.

- **Note:**

  This variable is set inside the include file GETVIDEO.BAS.

GPDat%(78) : Active Menu Color

- **Data Type:**

  Integer

- **Possible Values:**

  Possible values include color values which are designed for the GPrint0VE routine. The formula for combining a foreground color and a background color is

  ```
  GColor = Foreground + 256 * Background
  ```

- **Uses:**

  The routines PullDownG, VertMenuG, and MsgBoxG use this color to determine colors on the screen under the right circumstances.

- **Note:**

  This variable is set inside the include file GETVIDEO.BAS.

GPDat%(79) : InActive Menu Color

- **Data Type:**

  Integer

- **Possible Values:**

  Possible values include color values which are designed for the GPrint0VE routine. The formula for combining a foreground color and a background color is

  ```
  GColor = Foreground + 256 * Background
  ```

- **Uses:**

  The routines PullDownG, VertMenuG, and MsgBoxG use this color to determine colors on the screen under the right circumstances.

- **Note:**

  This variable is set inside the include file GETVIDEO.BAS.

GPDat%(80) : Normal Screen Color

- **Data Type:**

  Integer

- **Possible Values:**

  Possible values include color values which are designed for the GPrint0VE routine. The formula for combining a foreground color and a background color is

  ```
  GColor = Foreground + 256 * Background
  ```

- **Uses:**

  The routines PullDownG, VertMenuG, and MsgBoxG use this color to determine colors on the screen under the right circumstances.

- **Note:**

  This variable is set inside the include file GETVIDEO.BAS.

GPDat%(81) : GW Fonts Available

- **Data Type:**

  Integer

- **Possible Values:**

  Depends on the number of fonts requested.

- **Uses:**

  When the Graphics Workshop Font Active is set using the routine SetGWFont, this variable is used to ensure that the font actually exists. If you want to have more than one Graphics Workshop vector font, set the variable MaxGWFonts% to the number of fonts desired, prior to the include file GWFONT.GW. This element affects only the Graphics Workshop Vector fonts.

- **Note:**

  This variable is set in the include file GWFONT.GW. The value will be 1, if no value has been placed in the variable MaxGWFonts%.

GPDat%(82) : GW Font Active

- **Data Type:**

  Integer

- **Possible Values:**

  Must be less than or equal to the number of fonts available. See GPDat%(81)

- **Uses:**

  This variable tells the routines which of the fonts to use when drawing a string. This element affects only the Graphics Workshop Vector fonts.

- **Note:**

  This variable is set by the routine SetGWFont.

GPDat%(83) : GW Horizontal Font Spacing

- **Data Type:**

  Integer

- **Possible Values:**

  This can be any non-negative number. A reasonable number is 5 (pixels).

- **Uses:**

  This variable sets the spacing left and right between letters. This element affects only the Graphics Workshop Vector fonts.

- **Note:**

  This variable is set by the routine SetGWSpacing, but is initialized to 5 pixels in GETVIDEO.BAS.

GPDat%(84) : GW Vertical Font Spacing

- **Data Type:**

  Integer

- **Possible Values:**

  This can be any non-negative number. A reasonable number is 5 (pixels).

- **Uses:**

  This variable sets the number of pixels to move before writing the next line of text beyond the descender of any of the lower case letters like 'g', 'y' and 'j'. This element affects only the Graphics Workshop Vector fonts.

- **Note:**

  This variable is set by the routine SetGWSpacing, but is initialized to 5 pixels in GETVIDEO.BAS.

GPDat%(85) : GW Overall Font Height

- **Data Type:**

  Integer

- **Possible Values:**

  This variable is set to the sum of GPDat%() element 84 and the Graphics Workshop Font's Maximum Height as determined by the LoadOutlineFont routine. The Graphics Workshop Font Height is stored in the array FontSize%().

- **Uses:**

  Many of the routines use this variable to determine spacing from one line to another. This element affects only the Graphics Workshop Vector fonts.

- **Note:**

  This variable is set by the routine SetGWFont and by the routine SetGWSpacing.

GPDat%(86) : Draw an Outline Only

- **Data Type:**

  Boolean

- **Possible Values:**

  -1 Draw Outline Only
  0 Fill Font In

- **Uses:**

  This function allows you to create an outline font, or by drawing an outline font over a filled in font, it allows you to create a font with an outline of a different color than the rest of the font.

- **Note:**

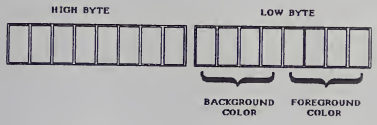
  This variable is initialized in GETVIDEO.BAS to 0.

APPENDIX D: CONVERTING FROM QUICKPAK OR GRAPHPAK PROFESSIONAL

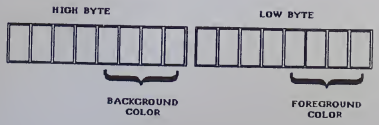
Many users from QuickPak Professional and GraphPak Professional are going to be using the features in Graphics Workshop. This Appendix is here to make the transition easier.

Converting from QuickPak Professional

One of the major differences between the two packages is the use of color. Because QuickPak Professional routines are designed for the text modes, the color values are optimized for that mode. Briefly, in text mode the foreground color is stored in the lower 4 bits of a single color byte, and the background color in the higher 4 bits.



Graphics memory is very different, because there is no explicit background color. There are many colors, and it is impossible to decide which of the colors at any one location on the screen should be considered the background color. Note that BASIC treats color 0 as the background color when in graphics mode. If you modify the background color for any location of the screen using the COLOR statement, the background color for the entire screen is changed. Graphics Workshop uses a system which supports different background colors at different regions of the screen. Rather than pack both colors into a single byte, the Graphics Workshop uses two bytes. Since an integer holds two bytes, this both faster and efficient. The figure below shows how these bytes are organized.



It is also faster to manipulate colors stored this way, because rotation of bits is not required to form the background color. All that is necessary is to isolate the high byte to obtain the background color.

In QuickPak Professional, a color for printing a text string is formulated with the following formula:

```
Colr% = Foreground% + Background% * 16
```

The Graphics Workshop formula is changed as follows:

```
Colr% = Foreground% + Background% * 256
```

Once the color values have been changed to work with the graphics modes, the next step is to find equivalent routines for those in QuickPak Professional. The following is a brief table comparing similar routines:

QuickPak Professional	Graphics Workshop
QPrintRC	GPrint0VE
ScrnSaveRest	GMove2VE
PaintBox	GPaintBox

The routine GPrint0VE requires a cursor position, a text string, and a combined foreground-background color value. Some of the QuickPak printing routines accept a row and column, while others do not. It is up to you to know where the cursor is located. You can use the CSRLIN and the POS(0) statements to find the current cursor location.

The GMove2VE routine stores and retrieves graphics regions using both the visible page and higher video memory. One important difference between this routine and the QuickPak equivalent is that GMove2VE uses the mixed coordinate system, while QuickPak only deals with text screen modes.

The GPaintBox routine emulates the PaintBox and MPaintBox routines as closely as possible. First, since text mode has only two colors for any character location of the screen, it is simple to specify what colors should be changed. In graphics mode it is possible for 16 colors to exist in a single character coordinate on the screen. The one important use for GPaintBox is to create a shadow effect in the pull-down and vertical menus. The primary difference is that only one color will be changed by this routine. In addition, the GPaintBox routine uses the mixed coordinate system.

Combining with GraphPak Professional

GraphPak Professional is already a graphics mode program, so fewer changes will be necessary. You should use the "standard" code shown in chapter 1 at the beginning of your programs. The one minor change is that instead of including GETVIDEO.BAS you should include SIMPLE.BAS. SIMPLE.BAS is the include file that came with GraphPak Professional and it contains more setup information which pertains only to GraphPak Professional. In your standard code, replace the line that says:

```
'$INCLUDE: 'GETVIDEO.BAS'
```

with

```
'$INCLUDE: 'SIMPLE.BAS'
```

You should also copy the GETVIDEO.BAS file from Graphics Workshop to your current directory. It performs many of the same functions as the one that comes with GraphPak Professional, but it handles more of the Graphics Workshop setup as well.

You must also use the COMMON.GW file instead of the COMMON.BAS file that comes with GraphPak. All of the variables contained in GraphPak's COMMON.BAS file are also contained in the COMMON.GW file.

The routines HercThere% and ScrnDump which were contained in GraphPak Professional are also a part of Graphics Workshop. HercThere% has remained unchanged, but ScrnDump has been greatly enhanced, and hence has been re-named to ScrnDump2. It has three added parameters which perform simple scaling, and tell the routine to print in either Landscape or Portrait mode.

Appendices

APPENDIX E: IMPROVING PIXEL ACCESS USING A CACHE BUFFER

The primary purpose of providing low-level graphics routines is to improve on BASIC's speed, as well as add capabilities. There are several factors which contribute to the slowness of manipulating graphics images. Perhaps most important is the need to access screen memory one pixel at a time. Another factor is the time required to pass and receive parameters, and yet another is the internal calculation to determine a pixel's address. Fortunately, EGA and VGA memory is organized in a manner that lends itself to a variety of speed-up techniques.

The Graphics Workshop uses a technique known as *cache buffering* to improve the speed of its low-level video routines, and it is described in this section. Please understand that the information contained here is presented solely for completeness, and you do not need to fully understand it to successfully use the Graphics Workshop.

A cache is an area of memory which is used to retain information being read from or written to other, slower memory. You may already be familiar with a disk cache, which serves a similar purpose. In that case, a cache reduces the number of times the disk drive must be physically accessed. By passing all data through the cache and remembering which data it contains, a cache routine can return the data from the cache instead of accessing the slower memory or hardware device. In the Graphics Workshop, a cache is used to avoid reading the relatively slow EGA and VGA memory when the same pixel or nearby values are accessed more than once. To appreciate how this cache is designed requires an understanding of how EGA and VGA memory is organized.

The first eight horizontal pixels at the upper-left corner of the screen are stored in a single byte at address 0. The next 8 pixels are at address 1, and so forth. To calculate the address of a particular pixel at location (X, Y) on the screen you simply multiply the Y value times 80, and divide the X value by 8. In assembly language a number can be divided by any multiple of 2 using bit-shifting operations, and this is must faster than a normal division. However, the multiplication is still necessary. If seven out of eight multiplications can be avoided, the pixels will be accessed faster than usual.

The GetCacheVE% routine remembers the last (X, Y) coordinate requested, the physical video memory address for that coordinate, and the data at that address when the request was made. Then, if future (X, Y) coordinates match the same physical screen address, GetCacheVE%

retrieves the data from its cache, without having to calculate an address or access the slower video memory.

Since it is possible to circumvent the cache routine, the data in the cache buffer could become obsolete. This can happen when a non-Graphics Workshop routine writes directly to screen memory without going through the cache. The routines DrawPointVE, DrawPointVEOpts, LineVE, and CircleVE, all reset a flag within in the cache routine, so GetCacheVE% will know that its data is not current.

Another way that the cache data could become invalid is by using a BASIC routine such as LINE or DRAW. Since these statements are not aware of our cache, they will change the screen but without updating the cache data or even clearing the flag. Although we assume you will use Graphics Workshop routines for all drawing, the ResetCache routine is provided to let you reset the cache manually if necessary. ResetCache would therefore be called after using the BASIC LINE or CIRCLE command, but before using GetCacheVE%.

We have also included the ReDrawVE routine, which lets you redraw the last pixel that was plotted. Unlike the other Graphics Workshop plotting routines, ReDrawVE does not require a pixel location. Rather, it simply redraws the most recently accessed pixel, based on the data currently in the cache buffer. This technique is shown in the comments that accompany ReDrawVE. Since it takes advantage of the cache, you must call GetCacheVE% to specify the most recently accessed location before calling ReDrawVE.

GetCacheVE%

Assembler
function contained in GW.LIB

■ **Purpose:**

GetCacheVE% is similar to GetPointVE%, except it uses a cache to operate more quickly. GetCacheVE% returns the color of the pixel at a specified (X, Y) coordinate.

■ **Syntax:**

```
V% = GetCacheVE%(BYVAL XPos%, BYVAL YPos%)
```

■ **Where:**

XPos% and **YPos%** specify the (X, Y) coordinate.

V% will receive a color value between 0 and 15 for the high-resolution EGA and VGA screen modes.

Comments:

All parameters for this routine are passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The routines DrawByteVE and DrawPointVE reset the cache so GetCacheVE% will return the correct value. If you use any other routine which modifies the pixels on the screen, use ResetCache before calling GetCacheVE% again.

A special routine, ReDrawVE, can be used to change the color of the point whose value was just determined. ReDrawVE does not require you to specify the pixel to change as it assumes the last position specified by the GetCacheVE% function.

■ **See Also:**

DrawByteVE, DrawPointVE, GetPointVE%, ReDrawVE, ResetCache

ReDrawVE

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

ReDrawVE changes the color of the last point returned by the GetCacheVE% routine. It utilizes GetCacheVE's cache to remember the last location accessed on the screen.

■ **Syntax:**

```
CALL ReDrawVE(BYVAL PointColor%)
```

■ **Where:**

PointColor% is a color between 0 and 15.

Comments:

The parameter for this routine is passed by value to provide the maximum speed. Including the file GWDECL.BAS at the beginning of all programs or modules which use this routine will insure proper operation.

The function GetCacheVE% must be called prior to using this routine. This routine, in conjunction with the GetCacheVE% function, can modify the color of any region of the screen with great speed. It does this by eliminating extra parameters and the need to re-calculate the screen position of the point. For example, to change all occurrences of the color black to red within a specified region use:

```
FOR s% = 100 TO 150
  FOR t% = 100 TO 150
    V% = GetCacheVE%(s%,t%)
    IF V% = 0 THEN V% = 4
    CALL ReDrawVE(V%)
  NEXT
NEXT
```

■ **See Also:**

GetCacheVE%, DrawPointVE

ResetCache

Assembler
subroutine contained in GW.LIB

■ **Purpose:**

ResetCache is used to refresh the cache used by GetCacheVE%. If you use a non-Graphics Workshop routine between two calls to the GetCacheVE% function, then you must call ResetCache between them to ensure that the GetCacheVE% cache is up-to-date.

■ **Syntax:**

```
CALL ResetCache
```

Comments:

If you make a call to the GetCacheVE% function, and then use the BASIC PSET statement to write over that pixel, and once again call the GetCacheVE% routine to ask it what the color is, GetCacheVE% will return the original value and not the value set by the PSET statement. This is because the PSET statement has no connection with the GetCacheVE% cache, and does not update it. If you call ResetCache after the PSET statement, then a following GetCacheVE% statement will return the proper value.

It is unlikely that you will need to call the ResetCache routine, but it exists for compatibility with BASIC graphics statements. A call to the DrawPointVE, DrawByteVE, LineVE, or CircleVE routines will also reset the cache.

■ **See Also:**

DrawPointVE, DrawByteVE, GetCacheVE%, LineVE, CircleVE

Appendices